JOLTS: Checkpointing and Coordination in Grid Systems

A Thesis Submitted to the College of

Graduate Studies and Research

in Partial Fulfillment of the Requirements

For the Degree of Masters of Science

in the Department of Computer Science

University of Saskatchewan

Saskatoon, Saskatchewan

by

Jeremy Pfeifer

Permission To Use

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
S7N 5A9

Abstract

The need for increased computational power is growing faster than our ability to produce faster computers. Already researchers are proposing systems that require peta-flop capable super computers, a far cry from what is currently capable. To meet such high computational requirements, networks of computers will be required. While it is possible to network together computers to achieve a single task, making that network more flexible to handle a multitude of different tasks is the promise of grid computing.

Grid systems are slowly appearing that are designed to run many independent tasks, and provide the ability for programs to migrate between machines before completion. However, these systems lack coordination capabilities. Many grid systems/environments allow multiple tasks to communicate/coordinate with each other based on various paradigms, but don't provide migration capabilities.

This thesis proposes a system, called JOLTS, that attempts to fill a gap by providing both checkpointing and coordination capabilities. The coordination model offered by JOLTS is based on the Objective Linda coordination language, with some additions. This thesis will show that the object space model is an effective form of coordination and communication, and can effectively be combined with checkpointing capabilities inside the same grid system.

Table Of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years there has been a large growth in the number of distributed client programs that the average computer user can download and run on their own computer to take part in large projects. The program with the largest following is *Seti@Home*, which processes satellite data to search for extra-terrestrial life. Other programs gaining popularity include *distributed.net* and *Folding@Home*. These programs can be classified as *distributed computing* programs because large computations are decomposed across many different machines. While these types of programs have many research benefits in their own right, they have also shown the type of computing power that can be accessed by networking common PCs together. Grid computing is concerned with how to effectively and efficiently network many computers together.

Processing power demands continue to rise, often faster than our ability to design and build faster computers. For example, researchers are trying to create teleimmersive environments for collaborative work [6]. However, the predicted processing power for such systems are in the peta-flop range, far more power than the world's faster super computer is capable of at 35.8 tera-flops [25]. Thus, the only way to achieve such requirements is to start networking together many computers. As will be shown, some grids stress the use of idle cycles on desktop computers (*see* Section 2.3.2), while others use dedicated hardware on the grid (*see* Section 2.1.4.1). Whatever method is used, the end goal of increasing processing power is the same.

1.1 Defining the Grid

The original use of the word "grid" in grid computing did not refer to the computers being networked together in a grid structure. In fact, it was used in reference to the power grid.

The idea was that eventually this single "grid" of network computers would be accessible from everywhere — it would be pervasive, much like the power grid is today. While this vision of the grid is no longer that common, it is still a goal of many grid researchers.

A more realistic view of grid computing is that of networking together computer *resources* to achieve a single goal. It is important to distinguish between computing resources and the entire computer. In some current grids it is possible to restrict what parts of a computer are accessible; e.g., a large disk may be used for grid storage purposes but it is not permitted to run jobs on the computer hosting this storage.

Grid computing was originally formed as an off-shoot from High Performance Computing (HPC) [6]. While it still shares many traits in common with HPC, unique properties inherent in grid systems have helped distinguish it from HPC (*see* Chapter 2). Because of this similarity, it isn't always clear what classifies as a grid system: cluster, super computer, or even a single personal computer. A cluster almost always consists of identically configured computers. These computers often reside in a single location, and the entire cluster is usually under the administrative control of a single person or entity. A super computer often takes a cluster to the next level; e.g., gigabit networking is deemed too slow and replaced by high speed connections such as InfiniBand. Gone are the days of a single computer; e.g., the Cray 3, being the most powerful computer. Most super computers today consist of a large number of commercial desktop computers networked together — though some also consist of large numbers of customized processors. For example, the 'Big Mac' at Virginia Tech, the third fastest computer in the world at 10.28 tera-flops is made up of 1100 Apple PowerMac G5s [25]. Super computers that are large clusters have some important additions. They usually contain high-speed network connections for fast communication between nodes, and special software for shared memory access.

The only real similarity between a grid system and a cluster/super computer is the fact that many computers are networked together to work on a common problem. In fact, a cluster or a super computer can be treated as a *single* node in a grid system, in grid applications referred to as *distributed supercomputing* (*see* Section 1.2). Nodes, or groups of nodes, in a grid system are usually under different administrative control. Also, nodes are usually different types of computers, sharing different types of resources, and running different operating systems. Some nodes on the grid may have special hardware, such as the high-speed connections found in super computers, but it isn't a requirement. Nodes

in a grid can also be geographically scattered, so high-speed and possibly even reliable network connections amongst nodes isn't always a guarantee. Whether or not a node is part of a grid usually depends on if it is running the appropriate software or not. It is possible to have an entire grid system running on a single computer: the server, worker, and client all running on a single computer.

1.2 Grid Applications

Grid systems are usually classified by the applications they run [6]. The grid community uses the term application to refer to the *type of problems* that a grid system is designed to handle. While no grid system is ideal for all situations, making adjustments to the system to increase performance in one area can result in a decrease in another. These tradeoffs result in configurations that fall into one of the following five categories:

1. Distributed supercomputing
2. High-throughput computing
3. On-demand computing
4. Data-intensive computing
5. Collaborative computing

Each of these areas will now be discussed in turn.

1.2.1 Distributed Supercomputing

Grid systems in this category are characterized by two properties:

1. The large size of the problems they are attempting to solve; and
2. The large amount of resources being thrown at the problem; e.g., several tera-flop super computers and multi-terabyte file servers.

Some systems qualify by both properties. Most grid systems in this category qualify by the second property, the amount of resources being used. Usually, several actual super computers are working as nodes in the grid. Another option is if most of the computers

in a very large company are used to work on the same problems. The computers can even be the personal computers on all the employee's desks scattered across multiple buildings. An example matching the first property are problems where the amount of communication between nodes is high. If intercommunication is low, the system might in fact be a high-throughput computing system.

1.2.2 High-Throughput Computing

High-throughput computing grid systems are characterized by the requirement to process as many data blocks as possible, as fast as possible. Computational problems that are easily decomposed into small pieces are ideal for this type of grid system. The communication between nodes working on various pieces are usually kept to a minimum — if any communication exists at all. Depending on the amount of resources applied to the problem, a grid could qualify as a distributed supercomputer; however, the fact that intercommunication is extremely low is what causes it to fall into the area of high-throughput computing instead of distributed supercomputing. An example of this type of system is Condor, which is discussed in Section 2.3.2.

1.2.3 On-Demand Computing

On-demand computing is characterized by having a need for large amounts of computing power, but only for a relatively short period of time. Users of this type of grid are usually more cost-performance driven than pure performance driven. This area holds lots of potential for e-commerce applications. For example, a company with a large grid system could sell (or trade) some of it's idle computing power to a second company that requires more processing power than it has available for a limited time to complete some urgent task. An example of such a type of problem is the scheduling problem at a university. It requires a lot of power, but is only performed periodically, maybe only twice a year. An example of an on-demand computing system is NetSolve [13].

1.2.4 Data-Intensive Computing

Data-intensive systems are characterized by the creation and processing of *large* amounts of data, stored both locally and remotely. These types of systems usually have high band-

width and I/O requirements, along with high processing power requirements to analyze the data. An example of this type of grid is one used to process the results from a particle accelerator, where the accelerator can potentially generate several terabytes worth of data each day. These large data sets can easily balloon into the petabyte range, requiring teraflop computers to process them. How to deal with such large data sets, such as moving, storing, and backing it up is an ongoing challenge.

1.2.5 Collaborative Computing

The word "collaborative" in collaborative computing does not refer to the fact that many computers are working together, it refers to the *users* collaborating toward some common goal. This is probably the newest area of grid computing and is still in its infancy. Since this area is heavily dependent on users, the systems in question need to be interactive. To achieve useful interactivity, i.e., real-time requirements, collaborative computing overlaps with distributed supercomputing, high-throughput computing, and data-intensive computing to some extent. Some of the systems theorized to fall into this category would require computers several orders of magnitude computationally higher than today's most powerful super computers.

An example would be a networking of several CAVE environments, scattered across the country, together where doctors could walk through a virtual body of a patient currently in a hospital imaging device. Thus, the data from the imaging device would qualify as data-intensive computing. Even to ship the relevant data across a network to various destinations is a challenge in and of itself. To do real-time rendering of this data might require a distributed supercomputing environment. Add to this the "collaborative" nature of the work; i.e., live video and audio streams between all the different CAVE environments so the users can interact with not only with their own CAVE, but with the users in remote CAVEs, and you have a very complex system. All of these requirements must be met in real time to effectively create a collaborative environment.

1.3 Thesis Goals

Of the five grid applications just discussed, this thesis is only concerned with high-throughput computing. As mentioned in Section 1.2.2 communication among nodes in

a high-throughput system is very low, if it even exists at all, which is the case of Condor, discussed in Section 2.3.2.

As mentioned earlier a high-throughput grid is concerned with processing data blocks as fast as possible. For long run jobs, it is possible that the number of nodes in the grid can be dynamic, increasing and decreasing over time. When a new node enters the grid, data blocks can be assigned to this node for processing. However, when a node that was processing a data block leaves the grid, the data block will either be lost entirely, or be restarted from the beginning. Another option that is just beginning to appear in the grid community is that of checkpointing. If a data block is checkpointed, when a node leaves the grid, a different node can continue processing the same data block from where the data block was when the checkpoint was created.

The combination of communication/coordination between nodes and checkpointing in a high-throughput grid is as non-existent research area. This thesis examines a high-throughput system developed, called JOLTS, that supports both a communication model between the nodes in the system based on Objective Linda (*see* Section 2.2.5), and checkpoints. The *high-level* design of JOLTS is described in Chapter 3, while the *detailed* design of JOLTS is given in Chapter 4. Because high-throughput grids are concerned with processing data blocks as fast as possible, the performance penalty of using the system, including its checkpointing capability, are examined in Chapter 5. Lastly, this thesis's research contributions and future work are given in Chapter 6.

Chapter 2

Grid Research

The area of grid research ranges from the theoretical such as examining what features are necessary in a grid system, to the practical such as implementing detailed protocols across a network. This chapter is divided into three main areas. Section 2.1 briefly describes some of the major design concerns for grid systems, such as scheduling mechanisms. Section 2.2 describes coordination languages, a special set of programming languages that can used to write programs designed for grid systems. Lastly, Section 2.3 describes a few grid systems that are similar to the JOLTS system, created as part of this thesis research, discussed in Chapter 4.

2.1 Research Areas

There are few, if any, fully-functional grid systems available today. As seen in Chapter 1, there are many categories of grid systems. Each of these system types focus on meeting particular requirements. Some requirements can be simple to solve, while others have no adequate solution. This section outlines some of the more active grid research areas.

2.1.1 Security

Security in a grid system is more than simply making sure all communication is encrypted. When a grid system is set up that allows outsiders to access resources available on various computers, security is a large concern. Many of the security concerns in grid systems are the same as most online systems; e.g., verifying user names and passwords. However, the distributed nature of grid systems often adds complications to the standard security concerns. Other areas are rather unique to grid systems; e.g., protecting resources from

malicious code. Other areas such as accounting, used when payment is required to use grid services, are closely coupled with security, and are beyond the scope of this thesis and will not be discussed.

2.1.1.1 Authentication and Authorization

Authentication is the process by which a user identifies him or herself to a computer system. This usually entails submitting some form of user ID and password to the system, which is checked against a list of known users and their passwords. This is a simplified view of how authentication is done in a non-grid system; e.g., logging onto a secure shell server. However, things get more complicated when a grid system that spans across several different administrative entities is involved.

A common problem in grid systems, when dealing with multiple administrative domains, is how to deal with a request for resources for a *principal* (a user or program entity) that isn't authenticated in some of the domains. For example, assume a principal submits a job to a grid system that uses some resources from domain a. During execution, the resources in a request — and receive — access to the resources in domain b on behalf of the principal. This works because the principal has a valid account in both domains. However, if the resources in b require access to resources in domain c, where the principal does *not* have an account, there is a problem. There are two main solutions to this problem: either always allow connections from other trusted grids, or use a trusted third party for all verification.

Trusting known grids creates the *illusion* of a transitive property. Using our previous example, if grid a needs access to resources in b, and b trusts a, represented as aTb, there is no problem. The same holds true for b requesting resources from c, where c trusts b, written as bTc. As mentioned at the end of the previous example, a going through b to use resources in c, is not a problem because c will be authenticating a request from b, not a. The original program running on a assumes the identity of a principal in b, this is known as *delegation of identity*. It may look like at transitive property of aTb and $bTc \Rightarrow aTc$; however, aTc doesn't hold true if c doesn't know about and trust a. Thus, there is the illusion of trust created by going through one (or more) intermediaries, represented as aIc. It is quite possible that $aTc = aIc$, but it is *not* guaranteed to hold for all pairs of grid systems. The end result of this relation is that it is possible for a single grid system to

access the resources of many other grids provided there is a path of trusted intermediaries in between.

The second option involves all the grid participants trusting the same third party. Requests made by a principal have a certificate attached. This certificate is then passed onto a third party that verifies the certificate. The third party can also be used to determine what the principal is allowed to do on a grid, called *authorization*.

Once a principal is authenticated, it is possible to determine its authorization level. This helps create different security levels, where different groups of principals are allowed to perform different tasks. As mentioned earlier, sometimes a principal will use identity delegation to change its identity. This can also be used to change its authorization level to perform a restricted task. This is similar to how a Unix process switches to kernel mode to execute kernel function calls. It is also possible for a principal to use *delegation of authorization* to allow some other process to perform tasks on its behalf. The same concepts, e.g., using a trusted third party, used to perform authentication can also be used to perform authorization.

2.1.1.2 Malicious Code/Hosts

Malicious code refers to programs that are run on a computer, designed to cause problems — very similar to a computer virus. A malicious host is the complete opposite, where the host computer intentionally modifies a program that it is running for malicious purposes; e.g., falsify output data.

Malicious code and malicious hosts have received relatively little attention in the grid community. One main reason for this neglect is the fact that most grid systems currently available are closed systems. Only authorized, *trusted* people are allowed to use them, and the computers on the grid are either dedicated computers, or trusted computers. This works fine in a controlled research environment, but if grid systems are ever to become easily accessible by anyone with an internet connection, the concept of malicious code and hosts is going to have to be dealt with.

Current grid systems simply have detailed logging systems, so that if anything goes wrong, it is possible to trace the cause. However, this is more relevant for finding malicious code, than malicious hosts. This is also more of a *reactive* approach, waiting until a problem occurs; opposed to a *proactive* approach where the problem is prevented from

even occurring. One proactive approach has been to restrict programs running on the grid to exist inside a *sandbox* environment. Programs inside a sandbox would have restrictions as to what they are allowed to do. Building a fully secure sandbox is a difficult problem; however, building from existing contained environments can make the job easier. The Java Virtual Machine (JVM) can be treated like a *weak* sandbox, since programs running inside the virtual machine have some restrictions. For example, objects inside the JVM aren't allowed to access low-level memory addresses. By adding new security managers, it is possible to make the security around the sandbox stronger, but at the cost of some functionality. This idea only addresses malicious code, it doesn't deal with the idea of malicious hosts.

As mentioned at the start of this section, malicious code is similar to a virus. As such, some techniques used to prevent a computer from becoming infected with a virus can be applied to malicious code. Malicious hosts are quite different, there is no repository of knowledge on how to deal with something like a hostile host. Once a program has been sent to a grid node for execution, it is very difficult to make sure the program isn't altered. This is because the node executing the program could potentially reverse engineer it to cause many types of damage. How to deal with malicious hosts depends on the nature of the program in question. For example, early versions of the Seti@Home client were altered by malicious hosts, allowing the owners of the malicious hosts to falsify their results to artificially inflate their completed jobs ranking [14]. The solution in this case was redundancy — a simple solution for a grid. Each data block processed was sent to multiple nodes, and the results of all the nodes processing that unique data block would have to be the same; otherwise, one (or more) of the results must have been falsified. Unfortunately, this solution will only work for a small class of problems. Only once more grids are easily accessible to the general public will grid security, especially dealing with malicious code and hosts, become a predominant research area.

2.1.2 Performance

Performance is concerned with maximizing available resource use, without overloading the resources causing a decrease in performance. A large portion of how to maximize resource usage is by trying to find optimal scheduling algorithms. Discussion of scheduling algorithms is deferred to Section 2.1.3.

Depending on the nature of the application being run, the CPU is often the most important resource. Specialized compilers can help keep the CPU busy to some extent, such as making sure all the Arithmetic Logic Units (ALU) are in use. The CPU can only be working if it has all the necessary data available for its current computation. To achieve this, many grid programs are starting to use asynchronous I/O, along with non-blocking function calls. These types of improvements are made by paying attention to the order of operations when writing a program. Consider the following example:

```
for(int i = 0; i < n; i++)
    sum += array[i];
previous_week = get_prev(); // synchronous network call
total = sum + previous_week;
```

This code fragment sums the values in an array, then fetches a previous value over the network, and finally adds them together to create a total. This seems like a natural way to program; however, the CPU is left idle waiting for data over the network. By using an asynchronous network call to request the data before it is first needed, the amount of idle CPU time can be reduced, or eliminated, depending on the network. The previous code fragment can be rewritten as:

```
previous_week = async_get_prev(); // asynchronous network call
for(int i = 0; i < n; i++)
    sum += array[i];
wait(previous_week); // check to see if the data is ready
total = sum + previous_week;
```

Now the program first fetches the data from the network. While this request is executing *concurrently*, the array can be summed. Depending on the array size and network conditions, the `wait` function would return immediately, keeping the CPU busy the entire time.

Another way to increase performance is to use efficient network protocols. Depending on the nature of the application, it might be possible to use UDP for speed, but most grid systems require reliable transmission so they use TCP/IP. Whether a program uses clear text-based messages, such as encoding them in XML or uses a binary protocol, can significantly affect performance. Using text-based messages has the advantage that they can easily be modified and understood. However, because of the extra characters these message can potentially take a longer time to transmit over a network. A binary protocol is more difficult to modify but is often more compact, requiring fewer packets to be transmitted; i.e., faster to transmit the complete message.

Disk access can also be an important factor in the overall performance of a system. As

mentioned earlier, asynchronous I/O can be used to help improve disk performance. In grid systems dealing with large data sets, it is sometimes advantageous to cache data for later retrieval. A fast disk (or RAID setup) can help reduce the time spent waiting for disk access. As networks continue to become faster, sometimes it is possible to retrieve cached data over the network faster than it is to retrieve the same data cached on a local disk. Of course, this only works if the remote machine has the data in memory; otherwise, the remote machine will have to use its local disk, in which case the original machine should use its local disk instead.

2.1.3 Scheduling

The initial scheduling algorithms used in grid systems were taken directly from massively parallel processor (MPP) schedulers. This seemed like a natural fit since grid systems consist of many processors that need to effectively communicate with each other. This didn't work very well because several of the assumptions made in MPP environments do not hold in grid environments, such as [6]:

- The MPP scheduler is in control of *all* resources.
- All the resources are under a single administrative domain.
- The number of resources in an MPP environment never change.
- Contention for resources from programs not under the control of the MPP scheduler is minimal.
- All similar types of resources, e.g., all disk resources, have the same characteristics.

All of these assumptions basically state that an MPP environment is a large, single entity, while a grid is made up of many smaller, diverse entities. Because of this diverse nature, grid schedulers are grouped together based on their desired goal. The following grouping of schedulers is *similar* to the groups of grid applications discussed in Section 1.2:

- Job schedulers — designed to work in high-throughput computing systems. The goal of these schedulers is to maximize the number of processed blocks in a given time span.

- Resource schedulers — designed to optimize the use of a limited resource in the grid. This can be in terms of keeping the resource busy as much as possible, or by ensuring fair access to the resource.

- Application schedulers — designed to increase the performance of a particular program running on a grid, at the expense of all other programs.

Job and resource schedulers are often grouped together as *grid schedulers*. What the grid will be used for determines what class of schedulers is ideal for the system. Even defining what "optimized" means for each scheduler can be a problem.

Creating a good, *customized* application scheduler starts by creating an accurate performance model of how the application behaves. When multiple programs are run simultaneously, the problem becomes harder, since the performance model will lose accuracy. When multiple applications are run simultaneously, it is virtually impossible to create an *optimal* schedule for any of the programs. Creating accurate performance models is difficult for several reasons:

1. Performance predictions must vary over time. The performance of the resources being used in the grid can vary according to time, which must be taken into account in the performance model.

2. Performance can vary based on the number and type of available resources. This information can be contained inside a dynamic profile, which should be incorporated into the performance model to give a better representation of the performance available for the resources.

3. If the program can be run in different environments, the performance model should be flexible enough to effectively model any possible environment it is going to run on, so an appropriate scheduler can be optimized for each environment. For example, having a fixed number of resources may not be an accurate representation of all the available environments.

Even something as simple as load balancing may not be good for the application, because it may cause an increase in communication overhead between nodes. Another problem with application schedulers is that a schedule which is good for one application, may not be good for the entire grid.

In the area of grid schedulers (specifically high-throughput computing systems), the current trend is to use advertisements and matchmakers to associate available resources with users/programs that need access to them [7]. These advertisements are often compared to classify ads in a newspaper, where people can advertise both items they have for sale, and items they wish to purchase. An entity with available resources publishes information, called an *offer*, regarding what resources are available to a matchmaker. Entities requesting resources also publish advertisements to the matchmaker, called *requests*. The matchmaker is then responsible for finding good matches between offers and requests. How to properly structure advertisements and what are good matchmaking algorithms is an active research area.

2.1.4 Communication/Coordination

All computers in a grid system need some form of communication. Whether it is as simple as transferring a single packet between a client and server, to advanced coordination issues carried out over a network between hundreds of nodes, communication is essential. For applications that run on a grid — not to be confused with the so-called grid applications discussed in Section 1.2 — there are two fundamental ways for nodes to communicate with each other: by passing messages, and by using shared memory. Message passing is a more popular paradigm than shared memory because it allows easier communication between multiple processor architectures, and has a larger number of supporting applications and software tools. When coordination is involved, each of these communication forms has a complimentary coordination style; message passing uses control-driven coordination, and shared memory uses data-driven coordination. Another communication style called a tuple space, which is similar to shared memory and also uses data coordination, is discussed in Section 2.2.

2.1.4.1 Message Passing

Message passing has been known as Remote Procedure Call (RPC) for several decades. As just mentioned, message-passing communication performs coordination through a control-driven approach. This means that procedures and functions calls are made between two processes, whether they are local to a single machine, or hosted on different machines. Message passing can be done at the programming level, using features such as Java's

Remote Method Invocation (RMI), or .Net's remoting. If multiple languages are involved, CORBA can be used to create common interfaces that can be implemented in many different languages.

Java's RMI mechanism requires the programmer to implement a few special interfaces to indicate that his/her code is meant to operate over RMI. The compiler then generates special versions of the classes called *skeletons* and *stubs*. These classes handle the communication between the two computers involved in the method call. Also required is a special entity called a *registry*. The registry is responsible for keeping track of what classes on a particular machine are available for other processes to call. Other processes wishing to gain access to these classes for remote access, need to do so through this registry. Once the proper object is retrieved from the registry, the object can be treated like a local object, when in fact all method calls made to the object are executed by the remote object. There are several problems with using RMI:

1. The registry can become a bottleneck in the system.
2. The programmer must know what computer is hosting the registry beforehand.
3. It is difficult to dynamically find objects in the registry. It is simpler if you already know the name of the object you are looking for.
4. It is possible to having naming conflicts in the registry.
5. Communication between objects isn't very bandwidth efficient.
6. When a new remote connection arrives, a new thread is spawned. This is very inefficient if the object is handling many remote requests. Whether this is a problem or not, and to what degree, is highly dependent on the Java VM implementation.
7. The programmer is restricted to a Java-only environment.

Some of these problems, such as problem 6 can be removed by using CORBA instead of Java. The registry is replaced by an Object Request Broker (ORB) that adds many features over the registry. Programs are written according to a language-independent specification. The ORBs are then responsible for communicating on behalf of their respective objects to create the illusion of local objects. While objects are what the ORBs

deal with at a high-level, it isn't a requirement that the programming language used be an object-oriented language. Message passing can also be done at a higher level, using special environments to help link multiple machines. The two main, competing technologies are the Parallel Virtual Machine (PVM) and the Message Passing Interface (MPI), discussed in Sections 2.1.5.1 and 2.1.5.2, respectively.

2.1.4.2 Shared Memory

The other major form of communication between multiple processes is by *shared memory*. Depending on how the processes are related, there are three different levels of shared memory.

1. Thread Level Sharing — threads and processes all reside in the same memory address space.

2. Process Level Sharing — involves multiple process each with its own memory address space, and a designated shared memory area.

3. Distributed Shared Memory — involves multiple processes, with different processes located on different computers.

Processes/threads communicate with each other by placing data inside the shared memory. Coordination is usually done by relying on specific values for certain variables inside the shared memory. For example, one process may keep executing a loop while some flag is set to true, while a second process is responsible for flipping the flag to stop the loop in the first process from executing.

While shared memory is the fastest way to do inter-process communication (IPC), it does have some drawbacks. The main problem with shared memory is restricting access to the shared memory when necessary; e.g., to prevent reading a variable before the proper value is assigned to it. There are many different primitives for protecting shared memory, usually language or library based; e.g., Java's `synchronized` and semaphores in Solaris, respectively. However, these memory protection schemes are beyond the scope of this thesis. Another difficulty is coordinating multiple processes so they execute in a specific order. For example, preventing some function from executing in one process until a different

function in another process has finished executing is more difficult than in a message-passing environment. Shared memory is often very low-level, making a shared-memory space between different processor architectures very difficult. For example, translating between 32-bit versus 64-bit, and between little-endian versus big-endian architectures is problematic. Each of the three shared-memory levels will now be briefly discussed in turn.

Thread Level Sharing Shared memory in programs at this level is the easiest to work with because no extra work is required to create the shared memory. Threads by definition all share the same memory space; so depending on the language used, variables declared in specific areas can be shared among several threads. For example, threads in Java are able to share instance variables, while variables declared inside a method are local to only that thread. This same behavior can be achieved with processes instead of threads; e.g., by using the `vfork()` function in Unix. This creates a new process, but doesn't duplicate the active processes memory space. Normally this newly created process is then used to execute a separate program, preventing any memory access problems. However, if this new process doesn't execute a separate program, the end result is two processes sharing the same memory address space, allowing for communication using the shared variables.

Process Level Sharing Programs at this level consist of two or more processes that are executing in separate memory spaces on the same computer. Library calls to the operating system are responsible for setting up the shared-memory area, gaining access to shared memory created by a different process, and detaching from a shared-memory area [17]. Usually access to the shared memory is restricted to either read-only or read-write access, based on a per-process basis. For example, the process that created the shared-memory area could have read-write access, while permitting other processes to have read-only access. Once the shared-memory area has been created, communication and coordination is the same as thread-level sharing.

Distributed Shared Memory Programs at this level consist of two or more processes that are executing on different computers. The basic idea behind distributed shared memory (DSM) is to allow the programmer to manage memory as if it were all local to a single machine. DSM can be done using either hardware or software solutions.

> In hardware approaches, such as those implemented on the Stanford DASH, the HP/Convex Exemplar, and the SGI Origin, local cache misses initiate data transfers from remote memory if they are needed. Software schemes such as Shared Virtual Memory and TreadMarks rely on the paging mechanism in the operating system to transfer whole pages on demand between operating systems. [6]

The advantage of the hardware approach is that it is faster than the software approach, but has the disadvantage that it is usually restricted to a single architecture. The software approach has the advantage that it can be run on different hardware, provided the software has been ported to the desired architecture. The disadvantage of the software approach is there is a much higher time penalty than there is with the hardware approach. Regardless of the approach used, DSM systems still have not been able to scale as well as message-passing approaches.

2.1.5 Tools

The tools used for creating grid systems are diverse. Depending on who is being asked, some grid systems themselves are treated just as tools. This is *not* the approach taken in this thesis. A grid tool is defined as a piece of software or hardware that helps in the creation of applications that run in a grid environment; e.g., profilers, compilers, CASE tools, and so on. A large amount of effort (*see* Section 2.1.3) has been placed on developing tools that create performance models for creating schedulers. The remainder of this section gives an overview of some of the various tools used in concurrent and parallel systems.

People outside the field of concurrent systems assume that a compiler can take care of everything for the programmer. Initially, companies believed they could take their legacy code and run it through a parallelizing compiler to breathe new life into their old software. Parallelizing compilers are still relatively rare, and are not nearly sophisticated enough to carry out the previous example. One problem stems from the fact that a compiler cannot create parallelism that isn't inherent in a program to begin with. When the same variable is used concurrently, simply parallelizing a sequential program can be dangerous, creating nondeterministic behavior, or even deadlock. Some success has been made in creating parallelizing compilers for functional languages, since they don't use stored variables.

Each function in a series of calls can be executed in parallel. The disadvantage to this approach is that based on the outcome of some functions, others may not need to have been evaluated at all, unnecessarily wasting processor cycles.

To help create concurrent and distributed programs, new programming languages are being created. Some are merely extensions of existing languages with new libraries, such as High Performance FORTRAN (HPF). Others aren't really programming languages at all, but coordination languages, discussed in Section 2.2. Some programming languages are *dialects* of existing languages, such as Titanium [26].

Titanium is not an extension of Java, it merely uses it as a base language. The compiler produces native code, not bytecode. Some of the more interesting features of Titanium are local and global references, and zone-based memory management. Variables in Titanium can be declared as local to a process, or global to a program. Global variables are accessible to any process of a program, even on both shared-memory and distributed-memory architectures. Zone-based memory management is used for (de)allocating objects. When an object is created, it can be created for a specific zone. Instead of releasing an object, the entire zone is released. Reference counts to zones are used to prevent releasing a zone that still has active references, while not requiring reference counts to objects inside the zone.

Tools can also be used to help observe, and potentially debug, live concurrent programs. An example is a tuple space (*see* Section 2.2) visualization tool created by Paul Bercovitz as part of the work in [4]. This tool allows a programmer to watch the creation of tuples inside multiple tuple spaces, and see how the tuples move between several active processes as the tuples are created and consumed. This can be used as a visualization tool for beginners, so they understand how tuples migrate. It is also a valuable debugging tool, allowing a programmer to trace a program while it is running to check for errors.

Some tools are available that allow users to assemble pre-built or custom-built components into an application using a graphical interface. An example of such a system is the Linear System Analyzer (LSA) [8]. This particular system is used to help scientists solve complex systems of equations. Graphical representations of the various components are moved and assembled on screen, where various output ports are connected to corresponding input ports, even if the various components are executed on different computers. The designers of LSA view this process as very similar to the graphical creation of an integrated

circuit. Each module not only sends its output to another component, it also writes it to a file. Thus, when a component along a path changes, new data is only calculated from the starting component, not from the start of the entire path. A framework is provided to allow programmers to create custom components that aren't part of the standard package. A sequence of components can then be saved for later use, and executed again on different data sets. This type of tool allows non-programmers to create semi-custom programs that take advantage of the power of multiple computers to solve complex problems.

2.1.5.1 Parallel Virtual Machine

The Parallel Virtual Machine (PVM) was originally jointly created by the Oak Ridge National Laboratory, the University of Tennessee, Emory University, and Carnegie Mellon University.

> The overall goal of the PVM system is to enable such a collection of computers to be used cooperatively for concurrent or parallel computation. [9]

This collection can be comprised of heterogeneous computers. Programs that run inside this virtual machine consist of *tasks*. Tasks are capable of starting/stopping other tasks on any other computer in the virtual machine. They also have the ability to dynamically add/remove other computers in the virtual machine. All these abilities are accessed using standard PVM libraries. The main languages supported by PVM are C, C++ and FORTRAN. Because some computers may handle FORTRAN data types differently, all FORTRAN PVM library calls are to special stubs, which in turn call the corresponding C function. The PVM system is responsible for message passing, spawning processes, coordinating tasks, and modifying the virtual machine, if needed. It also does any necessary type conversion, e.g., from big-endian to little-endian when required.

The PVM system allows programs to decompose complex problems according to functional parallelism, data parallelism, or a combination of both. Functional parallelism is when each computer in the virtual machine is assigned a specific function. For example, one would be responsible for taking data, while another would be assigned the responsibility of processing that data because it had a specialized vector processor. A third machine with a graphical display could be responsible for displaying the results. Data parallelism is when each piece of a program works on a small section of the data, also referred to as

Single Program Multiple Data (SPMD).

Tasks in the PVM system are assigned unique IDs, called task IDs (TID). User programs are able to access the current TID, as well as the other TIDs that make up the entire program. These TIDs are used when messages are being sent and received between tasks. For example, messages are sent using the `pvm_send` procedure, where data is passed into the procedure, along with the TID of the destination task. At the receiving end, a task must make a call to `pvm_recv`, along with the TID of where the message is coming from. Thus, communication is tightly coupled between tasks based on their TIDs.

The PVM requires that each computer that is assigned to a specific virtual machine be running the PVM daemon `pvmd`. The user is then responsible for starting up the PVM application on at least one machine, where the master task starts executing. This master task is then responsible for creating remote tasks on other computers in the virtual machine. Remote tasks can also be created manually by the user, if needed. Since each PVM is created from computers selected by the user, it is possible for a single computer to be used by more than one PVM at a time.

One problem with the PVM system is that the program to be run needs to be separately compiled for each unique architecture of the computers comprising the virtual machine. The compiled binary form must then be copied to the appropriate place on each computer before the program can begin executing.

2.1.5.2 Message Passing Interface

Unlike PVM, the Message Passing Interface (MPI) standard was created by a Forum [5], consisting of large number of both corporate and academic institutions. The version of MPI discussed in this thesis is MPI-2 [22]. The original idea behind MPI was to take the best components of several competing technologies and combine them, as opposed to modifying one to meet the needs of the forum. The main goals of MPI are:

- portability across different machines, to the same extent that languages such as FORTRAN are portable; and
- the same code should run on any machine that has the MPI library.

While MPI is designed to run both on homogeneous and heterogeneous systems, it *isn't* a requirement that the version for such systems be compatible with each other. The MPI

description specifies the semantics of the system, allowing implementors to optimize the MPI system for a particular machine and architecture. If it is desired to use a heterogeneous system, the programmer must use an MPI version that supports heterogeneous systems.

Process groups in MPI are created using a *communicator*, which creates a communication *domain*. The communicator is passed as an argument to all library calls. The advantage of the communicator is both for (sub)group communication, and isolation of user programs that potentially share procedure and function names. It is possible to do both intra- and inter-domain communication using a groups communicator.

MPI is suitable for both multiple-instructions multiple-data (MIMD) programs, and single-instruction multiple-data (SIMD) programs. The main library is designed to work with C, C++, and FORTRAN. Similar to CORBA, parameters to the MPI functions are designated as either `in`, `out`, or `inout`, indicating where the variable is used for input, output, or both.

Just like PVM, MPI uses a separate command to indicate whether it is sending (`MPI_Send`), or receiving (`MPI_Recv`) data. Portable data types are available that can be sent between heterogeneous systems (and languages), provided the proper constructors are used. Most messages are on a one-to-one basis, where the sender indicates the recipient of the message. It is also possible to do broadcast type messages using `MPI_Bcast` which sends the arguments to all processes in the same group as the sender, including the sender itself.

MPI also has functions for performing non-blocking sending and receiving, using `MPI_Isend` and `MPI_Irecv`, respectively (the `I` is for immediate). Whenever non-blocking I/O is used, regardless of the system, methods must be available for testing whether the requested I/O function has completed. MPI has two functions for this purpose, `MPI_Wait` and `MPI_Test`, that are blocking and non-blocking testing functions, respectively. Similar functions also exist to test for completion in *groups* of non-blocking I/O calls.

Sending messages over a network usually involves placing the message data into a buffer, and then transferring the buffer to a socket for transmission. The reverse is true for receiving — incoming data is copied from the socket buffer into a buffer that is accessible to the user's program. Proper handling of these buffers can have a significant performance impact, which is why MPI allows a limited form of buffer control by desig-

nating a communication *mode*. The four modes available are:

1. Standard — it is up to the MPI system to determine if a message is buffered before it is sent. It is the default setting.

2. Buffered — the message to be sent is buffered, and can be sent (and possibly completed) regardless of whether a matching receive call has been made. The buffer for the message must be supplied by the user's program. This version is invoked using `MPI_Bsend`.

3. Synchronous — similar to buffered mode, except the completion of the send only occurs if a matching receive has at least started executing. Thus, it is guaranteed that both ends of the message are synchronized on the communication before either can move on. This version is invoked using `MPI_Ssend`.

4. Ready — a send in ready mode will only start if the matching receive call has *already* occurred. If this mode is used before the receive call is invoked, an error occurs. This version is invoked using `MPI_Rsend`.

All of these special sends can match with any of the multitude of receives that are part of the MPI library. There are also matching non-blocking versions of all these send methods.

PVM and MPI are often compared, since they appear very similar at first glance. There is some debate as to whether this is even a valid comparison [11], since both systems were designed with different goals in mind. Some of these differences are very subtle, and some are more implementation-specific differences rather than specification differences. However, from the point of the discussion here, PVM and MPI are very similar in that they are both message-passing based systems, as opposed to shared-memory systems.

2.2 Coordination Languages

This section describes a group of programming languages called *coordination languages*. While some coordination languages aren't true *programming* languages, most were designed to be an extension to existing programming languages that wanted to add high-level coordination capabilities. As mentioned in Section 2.1.4, coordination in parallel programs is either control-driven or data-driven.

Control-driven languages are characterized as having one path of execution explicitly tell another (or multiple) path of execution when to do something, and the receiving path of execution was awaiting the notification. Thus, communication between paths of execution occur at known locations and times. In this regard multiple paths of execution can be considered to be tightly coupled to each other. Examples of control-driven languages include Conic, Darwin/Regis, and Durra [18].

Data-driven languages are characterized as having multiple paths of execution watching for a specific state change in various variables. When the desired change occurs, the path of execution can then take the appropriate action. The variables being watched need to be accessible to multiple paths of execution, which implies some form of shared memory. Because coordination is done using the state of variables instead of explicit messages, multiple paths of execution can be considered to be loosely coupled. Examples of data-driven languages include Linda, LAURA, Ariadne, and Sonia [18]. The main coordination language of concern to this thesis is Linda; thus, the following subsections describe Linda and *some* of its various extensions in more detail.

2.2.1 Linda

As just mentioned, Linda isn't a true *programming* language, it was designed to be an extension to existing programming languages that want to add high-level coordination capabilities. One of the most common variants is C-Linda, but Linda features have also been added to FORTRAN, Scheme, and Modula-2 to name a few [18].

> If two processes need to communicate, they don't exchange messages or share a variable; instead the data producing process generates a new data object (called a tuple) and sets it adrift in a region called a tuple space. The receiver process may now access the tuple. [4]

Linda isn't limited to just two processes, any number of processes can use a tuple space to communicate and coordinate their activities. A tuple is a series of typed fields, usually consisting of strings and numbers. From a programming point of view, a tuple can be thought of as a simple record structure.

Standard Linda defines four primitives that can be performed on the tuple space: `out`, `in`, `rd`, and `eval`. The `out` statement is used for placing tuples into the tuple space; e.g.,

in C-Linda:

```
out("test string", 30, 1.676);
out(23, 1, -6.1);
out("John Smith", 4321);
```

These statements create corresponding tuples and place them in the tuple space. The `out` statement is non-blocking — execution returns immediately after each call. The two primitives `in` and `rd` are used to retrieve tuples from the tuple space. For example, statements to retrieve the previously entered tuples would be:

```
in("test string", ?i, 1.676);
rd(?x, 1, -6.1);
rd("John Smith", ?emp_id);
```

Tuples are retrieved by matching a template created from the arguments passed into `in` and `rd`. The question mark (?), acts as a wild card with the restriction that any wild-card matching value is of the appropriate type. Matching values are then copied into the variable. Assuming the variables `i`, `x`, and `emp_id` are all declared as integers, `i` will be assigned the value 30, `x` will be assigned the value 23, and `emp_id` will be assigned the value 4321. Both statements block until a tuple matching the template is found; if more than one matching tuple is found, one is arbitrarily chosen from the matches. The *difference* between `in` and `rd` is what is done with the matching tuple. The `in` statement *removes* the matching tuple from the tuple space, while `rd` makes a *copy* of the tuple, leaving the original in the tuple space.

The final Linda primitive is `eval`. This primitive is similar to `out`, except that it inserts *active tuples* into the tuple space instead of the passive tuples like `out`. Active tuples contain their own path of execution, usually as a separate process. When the active tuple has finished processing, it turns into a passive tuple and the result of the `eval` is placed into the tuple space. Consider the following statement:

```
eval("process 1", x, y, foo(x, y));
```

where the function `foo` can use other language-specific, or Linda operations. A separate process is created to execute the function `foo`, with the result of the function attached to the rest of the arguments to `eval` creating a new tuple, that is then placed in the tuple space.

Two other primitives that are sometimes included with Linda are `rdp` and `inp` (the `p` stands for predicate). These two primitives correspond to the primitives `rd` and `in`, respectively. The difference between these "p" primitives and the regular ones is that the

former are non-blocking. When invoked, they return a Boolean value indicating whether or not a tuple matching the given template was found. Because of the difficulty in implementing the non-blocking behavior, these two additional primitives are not part of the standard Linda primitives, and are only supported by certain Linda systems.

The advantage of the tuple space over message passing is that the sender and receiver are decoupled. The sender no longer needs to know where the receiver is, or even who the receiver is. The receiver isn't aware of where the sender is, or even who it is. This is advantageous for programs that can potentially have a large number of receivers. For example, in a producer/consumer application using message passing, the producer would need to know about all the consumers. Making sure each consumer is used can be a difficult problem. However, using a tuple space is much easier. The producer places the data into the tuple space, and any free consumer can simply look in the tuple space for new data.

The tuple space can also be used to emulate low-level coordination techniques like semaphores. One process can place a number of tuples into the tuple space that restrict access to a critical section. The act of grabbing a semaphore is done using `in`, which blocks if no matching tuple exists. When a process is done with the semaphore, it can be returned to the tuple space using `out`. Consider the following

```
for(int i = 0; i < n; i++)
    out("semaphore");
⋮
in("semaphore");
... // critical section
out("semaphore");
```

where the variable `n` is the number of semaphores to be created. The critical section is protected by a semaphore, implemented using a tuple. Semaphores protecting different critical sections can be assigned separate tuple items to help distinguish the semaphores.

This is the basis of the Linda coordination language. As stated earlier, it is designed to be an extension to existing programming languages. Just as any language is modified or extended to add new capabilities, such as Linda's tuple space, the same holds true for Linda itself. Some of the various Linda-like extension languages will now be discussed.

2.2.2 Piranha

Piranha [10] and its associated system of the same name is designed to use idle cycles across workstations on a network. Programs that run in Piranha are usually master/slave types, where one node places the tuples into the tuple space, while all the slaves are responsible for removing and processing the tuples from the tuple space. There are three main functions in Piranha: `feeder`, `piranha`, and `retreat`. The `feeder` function runs on the node that submitted the job, and is usually the master, creating tasks in the tuple space. The `piranha` function runs on the other nodes, executing the slaves. The `retreat` function is used when a user begins interactive work on a node, and halts the executing Piranha job. It is responsible for restoring the global state of the tuple space; how it does this is application dependent.

2.2.3 Bauhaus Linda

Instead of using tuples inside a tuple space, Bauhaus Linda uses *multisets*. The tuple space is no longer a flat structure, it is possible to add multisets to a specific multiset to create a hierarchy. To accomplish this, the `out`, `in`, and `rd` operations of Linda have been redone to work with multisets. Bauhaus Linda also adds several move operations to manipulate existing multisets. Consider the example taken from [18], lowercase letters are binary tuples, and capital letters are processes:

```
{a b b {x y Q} {w {z}} P}
```

If the process `P` executes `move{w}`, the resulting multiset will look like:

```
{a b b {x y Q} {w {z} P}}
```

Other functions exist for moving multisets up and down in the hierarchy.

2.2.4 Bonita

The Linda primitives are designed to provide *synchronous* access to a tuple space (ignoring the uncommon `inp` and `rdp`). Bonita [20] primitives are designed to provide *asynchronous* access to tuple spaces. Bonita also makes use of multiple tuple spaces, so normal Linda operations now require an argument referring to a specific tuple space. Several of the Bonita operations are overloaded, which are as follows:

```
rqid = dispatch(ts, tuple | [template, destructive | nondestructive])
rqid = dispatch_bulk(ts1, ts2, template, destructive | nondestructive])
arrived(rqid)
obtain(rqid)
```

The three main Linda primitives, `out`, `in`, and `rd` are all overloaded in the `dispatch` primitive. The variable `ts` refers to the tuple space in question. If the second argument is a tuple, that tuple is to be placed inside `ts`, similar to the Linda `out` primitive. If a template is given, it means a tuple is to be retrieved from `ts`, followed by a flag indicating if it is destructive (Linda `in`), or nondestructive (Linda `rd`). This primitive is non-blocking, and the `rqid` (request identifier) returned refers to the matched tuple.

The `dispatch_bulk` primitive is used to move tuples between two tuple spaces. Tuples that match the template are transferred from `ts1` to `ts2`. If the last argument is destructive, it means the tuples are *moved*; if it is nondestructive the tuples are *copied*. The `rqid` returned is then used in conjunction with other primitives to determine how many tuples where moved/copied.

The final two primitives `arrived` and `obtain` are used to determine if a tuple is available, and retrieve a reference to the tuple, respectively. The `arrived` primitive is non-blocking, instantly returning either true or false. The `obtain` primitive is blocking — it only returns with the specified tuple when the requested tuple is finally available.

Bonita makes no reference to the Linda primitive `eval`. It was determined that process creation should be the responsibility of the underlying environment to create processes, not Bonita itself. However, as noted in [20], primitives to control the creation of processes may be added in the future.

2.2.5 Objective Linda

Objective Linda was designed with the goal of qualifying as a open distributed system [15, 16]. The main concerns it deals with are:

1. Heterogeneous hardware — the computers in such a system can consist of various architectures.

2. Heterogeneous software — the computers in such a system can consist of various operating systems. These can also have different programming languages in use.

3. Heterogeneous configuration over time — number and types of computers involved

in such a system are dynamic, and can change at any time. Thus, software running on such a system must be able to appear and disappear on their own.

To meet some of these challenges, the creators of Objective Linda felt that the original Linda specification was too restrictive. Instead of using tuples, Objective Linda uses objects, instances of ADTs that are described in an language-independent notation called Object Interchange Language (OIL). While this language appears to rely on objects, because it relies heavily on ADT descriptions, it is possible to use a non-object-oriented language that can adequately implement ADTs to create OIL objects. Standard object-oriented languages form a hierarchy of objects, but this property isn't required. OIL objects are not *subtype* related, but instead follow *matching* relations. This follows the Linda idea of performing tuple matching.

Ideally, it would be possible to perform object matching by supplying an object and any required predicates to the standard Linda `in` and `rd` primitives. To accomplish this, so called *function objects* would be required to represent the predicates. Instead, the root of the Objective Linda hierarchy, `OIL_Object`, has a `match` function which takes a *template object*. It is then up to the programmer to define how two objects are supposed to match. The `OIL_Object` class also has an `evaluate` method as a starting point, when creating active objects.

Because of the heterogeneous configuration nature of open distributed systems, Objective Linda views the standard blocking primitives `in` and `rd` as limiting, because of potential disconnects when the network is changing. The non-blocking primitives `inp` and `rdp` are viewed as semantically incorrect. Instead of immediate failure when a matching tuple can't be found, these methods should instead mean a matching tuple can't be found *at the moment*. The Objective Linda solution to this problem is to add timeout counters to the `in` and `rd` primitives. Depending on the value of the timeout counter, it is possible to have them behave like the standard blocking primitives (timeout of infinity), or the non-blocking primitives (timeout of zero), and everything in between. To keep the appearance of consistency, timeout counters are also added to the `out` and `eval` primitives.

Another problem seen with the original Linda primitives, specifically with `in`, is the inability to remove multiple tuples at once. This is an important ability that is often required for certain synchronization problems. The solution chosen is to have the primi-

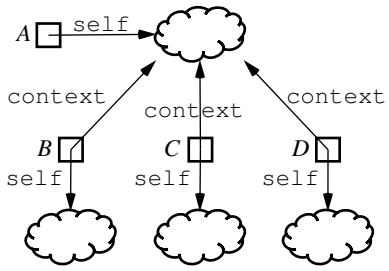

Figure 2.1: Multiple object spaces in Objective Linda

tives work on multisets. The Objective Linda multiset is treated as a container with three operations: `get`, `put`, and the current number of items `nbr_items`. With the addition of multisets, there is the need to specify the maximum and minimum values required in the multiset. For example, in a synchronization problem, it may desirable to grab either at least five, and up to 10 tokens at once, where 5 and 10 would be set as the minimum and maximum sizes of the multiset, respectively. Again, for consistency reasons, the other Linda primitives were augmented to deal with multisets and timeout counters.

An important distinction between standard Linda and Objective Linda are processes. As mentioned earlier, standard Linda treats both passive and active tuples as the same. When an active object, created from an `eval`, finishes executing it turns into a passive object. Objective Linda, on the other hand, has a clear distinction between objects and processes. Processes created as the result of an `eval` operation do *not* turn into a passive object when they finish, they simply disappear.

Objective Linda supports *multiple* object (tuple) spaces. However, it does not support *nested* object spaces; i.e., the class `Object_Space` is not a descendant of class `OIL_Object`. When a new process is created, it by default gets access to two objects spaces, one that is local to the newly created process called *self*, and one to the object space where `eval` was invoked on, called *context*. Of course, a process can create more object spaces, if needed. Consider Figure 2.1, if process A created three children: B, C, and D; each child would have access to their own private object space, as well as their parent's object space. While this may appear to form an object-space hierarchy, this is not the case as the private child object spaces are *not* stored inside the object space of A. Any communication between the children and their parent can be accomplished by placing/removing objects in A's object space. This communication is still limited, only processes that share the same parent can communicate with each other. The solution is to allow other processes access to the

private tuple space contained in each object. As mentioned earlier, objects spaces can't be nested, so instead of the children placing their `self` object space into their `context` object space, they place what is called an object space *logical*. The class `OS_Logical` is a descendant of `OIL_Object`, so it can be placed inside an object space. Once a *logical* has been placed in an object space, any process can use either `in` or `rd` to access it. Once a process has a different process's *logical*, it can then use the Objective Linda primitive `attach` to gain access to the actual object space. Thus, to summarize, the Objective Linda primitives are:

```
boolean out(Multiset *m, double timeout)
boolean eval(Multiset *m, double timeout)
Multiset *in(OIL_Object *o, int min, int max, double timeout)
Multiset *rd(OIL_Object *o, int min, int max, double timeout)
Object_Space attach(OS_Logical *o, double timeout)
int infinite_matches // constant for min and max values
double infinite_time // constant for infinite timeout delay
```

Using this set of primitives, the creators of Objective Linda feel that they have all the components necessary to qualify as an open distributed system.

2.3 Grid Systems

This section is meant to give a brief overview of two prominent grid systems currently available: Globus and Condor. The Globus system is designed around the idea of making services available to users on the grid. Condor is a high-throughput computing system designed to harness unused cycles on desktop computers.

2.3.1 Globus

No discussion of grid environments is really complete without mentioning Globus. The group behind the Globus project view the grid as a pervasive entity, much like the current Internet; however, it isn't meant to replace the Internet. Globus is not a true grid system — more accurately, it is a toolkit to help developers create grid-enabled applications. Globus is designed to create a grid software infrastructure of basic grid services. The Globus toolkit consists of seven essential services [6]:

1. GRAM — resource allocation and process management

2. Nexus — communication services, both unicast and multicast

3. MDS — information services, distributed access to structure and state information
4. GSI — security services
5. HBM — system health and status services
6. GASS — remote data access, providing both sequential and parallel interfaces
7. GEM — executable management (construction, caching, and location)

The most important of these services is probably the Metacomputing Directory Service (MDS). The grid infrastructure can change very fast, with new nodes coming and going, new services replacing old ones, and new services coming into existence. Applications will need to use this information service to keep track of all these changes. As applications use more of these services, it is deemed to be more "grid aware."

Services are deemed either as being *local* or *global*. Local services are very simple, so they can be deployed easily. Global services are more complex, but are built on top of the local services. The seven core services listed act as interfaces between the local and global services. Interfaces are usually designed to hide low-level details from outside interfaces; however, this is not the case in Globus. Globus interfaces are deemed to be *translucent*, allowing outside entities to gain information about the low-level services offered. The reason for this design decision is it gives the higher-level services the ability to fine tune the lower-level services to their particular needs.

Default implementations of many of these services are supplied to allow developers to get up and running with the Globus toolkit fast. Services can then be extended and customized as they see fit.

2.3.2 Condor

The Condor system, developed by the Condor Team at the University of Wisconsin-Madison, has been in use for more than 10 years [24]. Condor can be classified either as a batch system, or a high-throughput computing system. It is designed to allow users to submit jobs to a group of computers, called a *Condor pool*, where it is executed. The results of the job are returned to the user upon its completion. Two of the capabilities of Condor relevant to this thesis are its checkpoint and migration, and remote system calls

features. Before these concepts are discussed in detail, a high-level view of the Condor system is given.

Scheduling in Condor is done using an advertisement and matchmaking system as discussed in Section 2.1.3 on page 14, which Condor calls *ClassAds*. When submitting a job to a pool, the user must specify the requirements and preferences (called a *rank*) for the job, creating a ClassAd *request*. For example, the desired machine *must* have a minimum 128MB of RAM, and it is *desirable* to have the fastest integer-based performance possible. Computers also advertise their available resources when they join a Condor pool, creating a ClassAd *offer*. Some information is static, such as the processor and operating system, while other information is dynamic, such as the current load average. The Condor scheduler is then responsible for matching requests and offers, making sure all the desired *requirements* are met. In the case where multiple machines meet all the requirements, the rank comes into play.

The rank allows a user to specify either floating-point values or Boolean conditions to be met. A false Boolean value is given the value 0.0, while true is given a value of 1.0. Because a desired rank field may not be available on a particular machine, it is assigned a value of 0.0 for that rank. For example, if a program requests a machine with the highest floating-point performance, but none are specified in a computer's ClassAd offer, that computer is given a value of 0.0 for floating-point performance when performing a match. When multiple computers meet the requirements, the machine with the highest rank is selected. An example requirement and rank description is:

```
Requirements = Memory >= 512 && OpSys == "OSX"
Rank = cpus >= 2
```

This description says that the program *requires* a machine with at least 512MB or memory, running OSX, and it would *prefer* a machine with two or more processors. The `condor_status` command can be used to list machines that match specific constraints before submitting a job, to gain an idea of what information is available about the various machines in a pool.

When a job is submitted to a pool, it is accompanied by a description file. This file contains information about what files are used for input and output, the architecture required for the program, and so on. This file also contains the universe in which to run the program. Condor has several runtime environments, called a *universe*, for executing

code. The main ones of interest here are:

1. Standard Universe — jobs submitted to this universe need to be relinked to the Condor library using the `condor_compile` command. This allows the job to take advantage of the checkpointing, migration, and remote system calls offered by the Condor system.

2. Vanilla Universe — jobs submitted to this universe cannot be relinked to the Condor library. Access to files must be done using either a shared-file system, or through Condor's file-transfer mechanism. Fewer services are provided than in the standard universe, but at the same time fewer restrictions are placed on the job. This universe is used mainly when the object code isn't available, preventing it from being relinked and placed in the standard universe; e.g., commercial programs and interpreted languages.

3. PVM Universe — jobs submitted to this universe are written for the the PVM environment, as discussed in Section 2.1.5.1. This universe is designed for PVM applications that follow the master/worker paradigm of parallel programs. The machine that submits the original job to Condor is designated as the master, and cannot be preempted. Requests to add more machines to the PVM are transferred to Condor, which is responsible for finding an available machine. If a PVM node disappears, it's assigned task is restarted (not resumed), on another node inside the PVM. Most PVM programs can run inside this universe with no modification.

4. MPI Universe — jobs submitted to this universe are written for the the MPI environment, as discussed in Section 2.1.5.2. Condor currently supports MPICH versions 122 through 124. Programs submitted to run in this universe will have to be compiled with `mpicc`. The machines used to execute the MPI program are required to be setup as *dedicated* resources; i.e., the jobs will not be preempted or suspended if a user sits down at the keyboard.

5. Globus Universe — jobs submitted to this universe are written for the Globus system, as discussed in Section 2.3.1, but are submitted through Condor. Job submission is done through the GRAM protocol. The advantage of using Condor to submit jobs to

Globus is that Condor helps deal with common errors that can arise with executing a job, even if it is executing in the Globus system.

6. Java Universe — used for jobs written in Java. Jobs in this universe have no checkpointing or remote systems calls abilities. Jobs can be submitted either as a single class file, or as a jar file.

Checkpointing in the Condor system is *only* available to jobs submitted to the standard universe. Checkpointing works by using custom C library calls to save the entire memory footprint and execution stack of a program, similar to a core dump. The saved form is then transferred either to the computer that originally submitted the job to the pool, or to another machine designated by the user as a *checkpoint server*. The system periodically checkpoints an executing job *automatically*. The job can also create checkpoints itself by calling one of two library routines:

```
ckpt() // performs a checkpoint and returns
ckpt_and_exit() // checkpoint and the exits
```

The second version causes the job to halt entirely, but can be restarted later by the system. While the original C/C++ program submitted may have been platform independent, the resulting checkpoints are not. For example, a program that started on a Linux machine using an Intel architecture can only use that configuration from then on. It cannot be resumed on another machine; e.g., a Solaris machine with a UltraSparc processor, even if the original program runs on that configuration. While the Condor system runs on many different architectures and operating systems, only a small subset support the *standard* universe.

Migration is tied very closely to checkpointing. Migration is when a job needs to leave the machine it is currently executing on for various reasons; e.g., it is now being used locally. When this happens, there are several options on what to do with currently executing jobs:

1. The job can be checkpointed and resumed somewhere else (standard universe).

2. The job can be stopped and restarted from the beginning somewhere else (vanilla universe).

3. The job can be temporarily suspended until the machine becomes available again (vanilla universe).

Of course, option 3 isn't viable if the machine in question crashed, in which case option 2 must be used. While the idea of checkpointing and migration seem like a great idea, using these capabilities results in some restrictions being placed on programs in the standard universe [24], including but not limited to:

- Multi-process jobs are not allowed, including system calls `fork()` and `exec()`.
- Inter-process communication is not allowed, including pipes, semaphores, and shared memory.
- All network communication must be *brief*, because it can delay checkpointing and migration.
- Alarms, timers, and sleeping are not allowed.
- Opened files must either read-only or write-only. A file in read-write mode will generate a warning, but not an error; however, it can cause significant problems if the program is resumed from a checkpoint.
- Memory mapped files are not allowed.
- Multiple kernel threads are not allowed; however, multiple user threads *are* allowed.
- A large amount of disk space is required for the checkpoints, from 10M and up. This can be alleviated by designating a *checkpoint server* in the description file.

The major advantage of the standard universe, besides checkpointing and migration, are the remote system calls. When a system call is made to a file in the standard universe, the Condor system intercepts the call and redirects it to the machine that submitted the job to the pool. When a job is submitted to a Condor pool, a special process called a *condor_shadow* is started on the user's machine. This process receives any systems calls from the submitted job, and sends the results back to the remote machine in the pool. The net effect is that the program can remain unmodified from running on its originating machine to running inside a pool. The disadvantage of this setup is that the originating machine must remain on and connected to the network while it's job(s) remain unfinished.

2.4 Summary

With all the active research areas in grid computing, it is somewhat surprising that there aren't more grid systems currently available. Often these research concepts and examined in isolation, without the existence of a full grid system to test them. Absent from the major research areas discussed in Section 2.1 is checkpointing. While checkpointing isn't necessarily a valuable feature for all grid applications (*see* Section 1.2), it can be beneficial for high-throughput grid applications.

The most common high-throughput grid system with checkpoints is Condor, discussed in Section 2.3.2. While it has many useful features, it has a serious limitation in that jobs are restricted to a single computer. There is a hole in grid research — there is no data available about the interaction between checkpointing and coordination issues. The JOLTS system, described in the following chapter, is designed to investigate the problem of combining checkpointing capabilities in a grid system with a coordination model that allows jobs to communicate and coordinate while existing on various computers. The coordination model used in JOLTS is an extension of Objective Linda, described in Section 2.2.5.

Chapter 3

JOLTS High-Level Design

Grid research is a large area, as shown in the number of topics covered in Chapter 2. One topic that wasn't mentioned in the grid research areas in Section 2.1 is checkpointing. While fault tolerance and recovery is a sub-area of performance, it is usually handled by detecting failures, and *restarting* the failed job. Only recently has checkpointing started to receive more attention, as researchers are realizing that restarting a job isn't always an acceptable option. For example, if a program has been running for six months, with one week of computation time left, what happens if a power failure occurs? With checkpointing, only the work since the last checkpoint was made is lost; e.g., two days, as compared to six months if the entire job had to be restarted. A few systems are starting to support checkpointing, such as Piranha and Condor, discussed in Sections 2.2.2 and 2.3.2, respectively. One problem with the current checkpointing systems is they are designed to operate independent tasks — they have no support for highly cooperative tasks. The reason is that resuming a small piece in a large parallel system is a non-trivial task. Thus, a new system was developed, called *JOLTS* (Java Objective Linda Tuple Space), to support checkpoints for coordinating jobs.

This chapter is meant to both describe some of the features of the JOLTS system, and give a description of some of the design *decisions* that went into its creation. Discussion of the *actual* high-level design is left to Chapter 4. Section 3.1 gives a conceptual overview of how the JOLTS system works. Section 3.2 describes the design goals for the JOLTS system. Section 3.3 describes the checkpointing system present in JOLTS. Section 3.4 and onward follows the ordering of grid research areas discussed in Section 2.1, and how they relate to JOLTS.

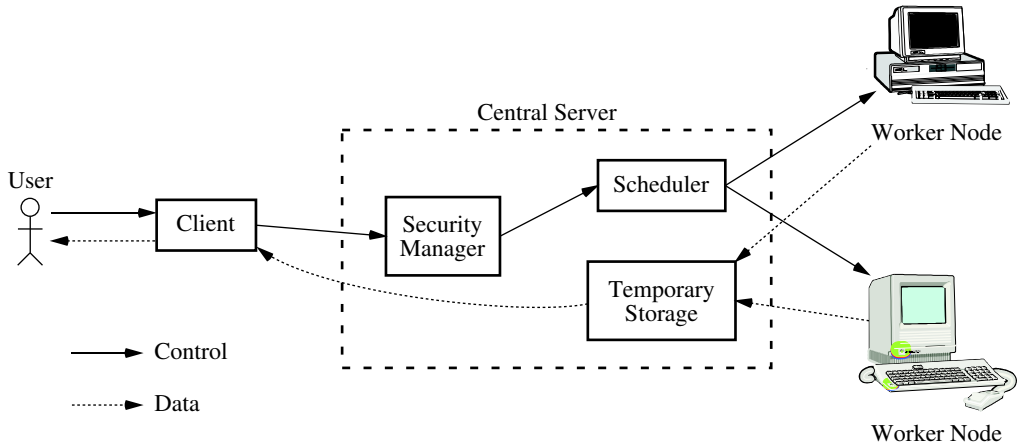

Figure 3.1: High-level conceptual diagram of JOLTS

3.1 Conceptual Overview

A high-level conceptual view of JOLTS is given in Figure 3.1. Using the grid "application" definitions discussed in Section 1.2, JOLTS is classified as a high-throughput grid. It is also *not* classified as a peer computing system, since it relies heavily on a central server and nodes in the grid don't talk directly to each other.

The main reason for choosing Java as the implementation language is the portability aspect, since it will run on any computer that has a JVM. This moves the potential number of nodes at the University into the hundreds (possibly thousands), instead of a few dozen, if it was restricted to Solaris, for example. Another reason for choosing Java is its threading capabilities. Any grid system is inherently highly concurrent, and without good thread (or process) support, implementing such a system would be a near impossible task. When designing the system, it is important to remember that users will be writing their own programs to run on the system. This implies implementing the system in a language that is familiar to the end users of the grid. Few users will want to learn a new language so they can run their programs on a grid. Also, porting existing programs to the system is more difficult if the language used for the program needs to be changed. Thus, the final reason JOLTS was implemented in Java is that many of the computer science students at the University of Saskatchewan are familiar with it, as opposed to HPF.

Users wishing to submit jobs to the grid system do so using the supplied client (graphical or command-line based), provided their program implements the proper interface (*see* Section 4.2). Programs need to be sent to the central server, either as a class file or a

jar file, along with any input files the programs will need. Once the job passes through the security checks (*see* Section 3.4), the server returns an ID that is assigned to the job. The user then requests the results for his/her program by returning the ID to the central server. The server will then return one of several messages based on the current state of the user's program; e.g., "try again later, your program is still running."

The worker nodes are where the programs submitted by the users are actually executed. When a worker node first joins the grid, it sends some basic statistics information to the central server; e.g., Java version, operating system, available memory, and so on. When a user job arrives at the central server, the server determines which worker node that job will be assigned to (*see* Section 3.6). The user's program is then transferred to the worker node, where it executes. When the job is finished, the results are sent back to the central server, where they are stored until the user requests their results. The results are in the form of Java objects, *not* text files.

3.2 System Goals

This thesis research focuses on how to effectively design and implement a checkpoint mechanism for highly parallel, coordinating programs running in a grid environment. This consists of designing and building a new grid system, with specific goals in mind:

1. The system should keep overhead to a minimum.
2. The system should make efficient use of the available resources.
3. The system should be scalable.
4. The system should support several different parallel/concurrent programming models.

It is generally well known that simply parallelizing a sequential program doesn't result in a linear speedup. For example, breaking a program into four pieces and running them in parallel doesn't automatically imply the result will be four times faster, there is overhead involved. Possibly the largest piece of overhead is moving the sequential program into a parallel form; however, how to do this is beyond the scope of this thesis. If a system is used to run the resulting parallel program, overhead exists in network communication, creating

the parallel processes, assembling the results, and so on. The first goal is to minimize the JOLTS overhead to keep it within some acceptable margin.

The second goal is straightforward, the grid system should not consume excessive resources, the majority of the resources on the worker nodes should be available to the programs running on the grid. The second goal will require careful design and programming of the grid system itself. For example, passive loops should be used instead of active loops when waiting for specific events to occur.

The third goal requires that the system should be able to handle large numbers of nodes and concurrent programs. Thorough stress testing will be required to ensure the scalability of the system. For example, a series of tests will be designed and run to determine the number of worker nodes that the central server can handle before performance is adversely affected. Another scalability issue is to determine how the performance of applications running on the grid scale, as more nodes are added to the grid. For more information about how the system was tested, refer to Chapter 5.

The final goal will be met using several levels of Application Programmer Interfaces (APIs), discussed in the following subsections.

3.2.1 Simple/Sequential API

The first API is a simple, beginners API designed to get programmers familiar with the system, how it works, and how to write programs for it. It has a method for receiving a mechanism that the programmer can use to create checkpoints (*see* Section 3.3), and an execute method. The execute method should contain the main body of the program to be executed. Since programs are actually objects, the constructor used to create the object/program should have no constructors. This API is also beneficial for programs that can't be parallelized, but can still benefit from using the checkpoint mechanism in JOLTS.

3.2.2 Parameter Experiments

In computer research experiments, programs are often run repeatedly, only varying the input parameters so results can be collected. These input parameters can be varied, from input data to flags indicating what algorithm is to be tested. For example, if different caching schemes are being tested in a network simulator, each run through the program can be to test a different caching algorithm, while the data that is being cached is identical

for each run. This type of parameter experiment is suited perfectly to a grid environment. Each node in the grid can be assigned a different parameter configuration for the same experiment. This approach has the advantage of greatly reducing the experimentation time from a sum of all the experiments, to the time required for the longest experiment (provided there are at least as many nodes as there are parameter configurations). Two different APIs are available in JOLTS to allow the running of parameter experiments.

The first API is for Single Instruction Multiple Data (SIMD) experiments. This type of experiment is characterized in that the exact same program is run repeatedly, but the data that is being processed in each run is different. Returning to the network simulator example, an SIMD example would be a simulator that is determining average request times, based on several different web server logs. Each node would be running the same program, but each node would be using a different input file.

The second API is for Multiple Instruction Single Data (MISD) experiments. This type of experiment is characterized in that slight modifications of the same program (or a different program) is run, but that the data being processed is identical for each program. This is the type of experiment described at the start of this subsection, where different caching algorithms are being tested, each using the same input file.

SIMD and MISD are very similar and can in fact be viewed as two different ways to attack the same problem. If the parameter experiment is designed to fill a table with experiment results, SIMD and MISD basically determine whether the results table is being assembled a row at a time, or a column at a time. The choice to use one or the other will depend both on personal preference, and how the programmer views the experiment he/she is conducting.

In addition to the Simple/Sequential API, the SIMD and MISD APIs require the implementation of a *collector*, used to assemble the results from the various nodes in the grid as their parameter experiments are completed. A default implementation is supplied as part of the API; however, the programmer is free to write his/her own collector from scratch, if desired.

3.2.3 Object Space

While the previous APIs have dealt with largely independent programs, this isn't an accurate reflection of the full capability of a grid system. To harness the full potential of a

grid system requires that nodes inside the grid have the ability to communicate with each other to accomplish a task. As mentioned in Section 2.1.4, there are two main ways to have components communicate with each other, by message-passing or using a shared-memory space.

Message passing appears to be the dominant form of communication in grid systems, but it relies on the fact that the remote system receiving the message is at a known location, and it knows about the sender; i.e., the sender and receiver are tightly coupled together. The ability of jobs in the JOLTS system to move between nodes in the grid breaks this assumption about message passing. There are several different ways to solve this problem, but each solution also has an associated problem. One common solution is to leave a forwarding pointer/address on the worker node, when a job moves to another worker node. The problem with this solution is that when code is moved to a different node because the original node crashed, that forwarding pointer will not be available. Another solution would be to leave a forwarding pointer on a central server. This approach has the advantage over the previous solution in that it is an assumed grid property that the central server will not go down; thus, the forwarding pointer will always be accessible. However, if the job that moved has not been assigned to a new node; i.e., there are no free nodes on the grid, the forwarding pointer will be empty. This implies that the message being sent needs to be stored until the moved job is assigned to a new node in the grid, and then the message can be forwarded. This solution does work, but instead of passing messages around, this design is much closer to the shared-memory space design.

Shared-memory spaces are usually for communication inside the same machine, but it is possible to create distributed shared-memory systems as discussed in Section 2.1.4.2. Ideally each node in the system would be able to share part of it's memory; however, moving code in the JOLTS system will cause serious problems. Because any node could potentially need to talk to any other node, the situation is worse than in message passing. Instead of having just sender/receiver coupling, all the nodes would be tightly coupled to each other. If any node crashed, it could potentially take with it an important piece of the shared-memory space, which is unacceptable. The solution is to use a centralized shared-memory space that doesn't move. This suggests using a tuple space, like those discussed in Section 2.2. The specific tuple space to be used in the system is based on the Objective Linda model discussed in Section 2.2.5. From a design point of view, the tuple

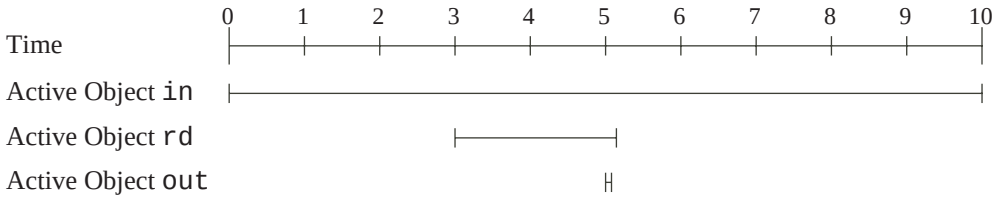

Figure 3.2: Timeline for example object space operations

space is similar to the Blackboard architectural style [3], with the blackboard located on a central server.

3.2.3.1 Object Space Clarification

One clarification to Objective Linda needs to be made before moving on. The Objective Linda specification only specifies *what* the available primitives do, not *how* they do it. A major ambiguity exists in how searches for matching objects in the object space are performed. When a single match is found, there are two options available:

1. Store a reference to the matching object in a temporary location, leaving a reference in the object space; i.e., no modification is made to the object space.

2. Store a reference to the matching object in a new location, and *remove* the reference in the object space; thus, no other active object will be able see/match the removed object.

Depending on which option is selected, it can have a dramatic effect on programs using the object space. Consider the following example, illustrated in Figure 3.2.

Assume the object space contains two instances of class `A`. One active object performs an `in` looking for instances of `A` with $min = 1$, $max = 5$, and $timeout = 10s$. The two matches are found instantly, but time remains to potentially find three more matches. After three seconds another active object arrives to perform a `rd` also looking for instances of `A` with $min = 1$, $max = 3$, and $timeout = 3s$. After an additional two seconds, a third active object arrives that inserts one instance of `A` using `out` and then leaves. The question then becomes, which of the remaining active objects (the `rd`, `in`, or both) see the newly inserted instance. JOLTS makes the decision that modifications to the object space are atomic, incremental modifications are *not* allowed; i.e., it uses option 1 given above. Thus, both active objects see the new insert. Hence, the active object performing the `rd` finds a

third match and returns because it has found it's maximum number of matches. The `in` thread waits an additional 5 seconds until it times out, at which point it returns the three matches it found and removes them from the object space. The results returned by both active objects are both predictable, and repeatable.

Another option would be to allow the active objects to temporarily remove matches as they find them; e.g., using option 2 given above. Using this idea, the `rd` in the above example will find either 1 or 0 matches, depending on if it sees the newly inserted item before the `in` active object does. The `rd` does *not* see the initial two objects, since the `in` temporarily removed them. Assume the `in` sees the insert first, causing the `rd` to find 0 matches. Thus, the returned results are 3 and 0 for the `in` and `rd` active objects, respectively. However, it could just as easily have been 2 and 1 for the results. The results are thus nondeterministic, which is usually to be avoided if possible in concurrent programs.

As can clearly be seen, the results found using the object space can vary depending on whether or not incremental modifications are allowed to the object space. As a final example using incremental modifications, change the value to $min = 5$ for the `in` active object. After ten seconds, when the `in` exits with zero matches, the object space will contain three instances of `A`. However, even though they were all present when the `rd` was active, *no* matches were found. This seems counter intuitive, and its why JOLTS doesn't allow threads to perform incremental modifications to the object space.

3.3 Checkpoints

Checkpoints in JOLTS are created on the worker nodes, but stored on the central server. Thus, if a job finishes prematurely on a worker node; e.g., the worker node crashed or the load became too high causing the job to be terminated prematurely, the server can resume the program from the most recent checkpoint on a different worker node. Checkpoints are associated with a number indicating how far along program execution has advanced. Using time stamps doesn't work given the fact that work nodes can have different performance levels, and that all the clocks on the worker node clocks would need to be synchronized for it to work.

The first decision in designing checkpoints into a grid system is to consider how the

checkpoints will be created. There are three main options to consider:

1. Allow the system to arbitrarily create a checkpoint when it needs to.

2. Allow the user to specify when a checkpoint should be made.

3. Both options 1 and 2.

Option 1 would appear to be the best option and is the one used in the Condor System (*see* Section 2.3.2). This option has the advantage that the user doesn't have to decide when to create a checkpoint of his program. This option can decrease execution time if the checkpoints are expensive to make and save for future use. The downside of this option is that if the system creates a checkpoint in the middle of a method, it can be very difficult, or impossible, to resume execution from that point if the program is restarted from a checkpoint file. This can be done in C using custom calls to manipulate the stack; however, this does not work in Java, which is JOLTS's host language. A solution to this problem is to use a customized JVM, as is done in the NOMADS system [23]. Using a customized JVM was dismissed because this JVM would then need to be ported to all the architectures that were going to host JOLTS worker nodes. Option 2, allowing the user to create their own checkpoints, is more feasible.

In the end, only the programmer knows when her program is in a safe state to create a checkpoint. The programmer is also in the best position to determine if there is any real use in creating a checkpoint for her program. For example, if the program is only going to take six minutes to execute, there may not be any benefit to using checkpoints, since it would not take too long to just restart the program from the beginning. On the other hand, if the program is going to run for six weeks, checkpoints could become invaluable.

To create a checkpoint, the user's program needs to call the `checkpoint(short)` method. The `short` passed in is just used to give ordering to the checkpoints, so the checkpoint archiving mechanism knows which one is the most advanced checkpoint. The downside of having the user indicate when to create checkpoints is that he might accidentally create them when his program is not in a stable state, making *proper* recovery from the checkpoint impossible. Regardless of which option is chosen, there are still some fundamental problems to be addressed with resuming from a checkpoint; namely, files and sockets.

File objects are references to files on the local computer. When a checkpoint is created and resumed on a different machine, any files associated with the program will not have moved with the program. However, the user should not be responsible for recreating file references each time a program is restarted from a checkpoint; thus, a special file is needed. Instead of working at the file level, the support class, `GridInputStream`, operates at the stream level. This class is designed to reconnect to the source file on the central server when it is deserialized; thus, from the program's point of view, the file *did* move with the program from one machine to the next.

A similar problem is faced when writing files, but file writing currently is not implemented in JOLTS. Any data that would normally be stored in an output file should instead be stored inside the object representing the job. The reason file writing hasn't been implemented is that any files that are written would need to be written on the server, not the worker node. This would be done for two main reasons:

1. If the worker node crashed, the output file would be lost.

2. Programs could potentially do malicious things to the worker node, if they had write access to the disk.

Writing to the server for sequential programs wouldn't be that hard. The challenge comes when any of the supported parallel APIs want to write to the same file, at the same time. Simply appending data to the end of the file could result in a scrambled file as thread switches take place on the server. There is no clear cut solution to this problem; thus, why file writing isn't supported in JOLTS. File writing is actually restricted in the default JOLTS security configuration to prevent this problem from even arising.

Similar to the problem of file references to files being lost when a program moves, socket connections are broken when a program moves as well. As mentioned earlier, the file problem is solved using a customized input stream to create the illusion of the file moving with the program. This technique will not work for sockets. The problem with moving a live socket connection from one machine to another is that the remote end of the socket, possibly on a machine not part of the grid system at all, would view this as a security violation. The solution to this problem is to create a *switchboard*, an intermediary between the program running on the grid, and remote end of the socket. When a program moves from one machine to another, it breaks its connection to the

switchboard, and then re-establishes the connection when the program is resumed on a different machine. The remote machine does not see any actual socket interruption, since it remained connected to the switchboard the entire time. Currently, JOLTS does not have a switchboard mechanism.

3.4 Security

While security is an important issue in large commercial grid systems, it is not an *overriding* concern in JOLTS. As seen in Section 2.1.1, a lot of security research focuses on authorization, authentication, and how it is handled when crossing administrative domains. In JOLTS, security focuses more on the concept of malicious code, preventing programs submitted to the grid system from doing damage to nodes they execute on in the grid.

The built-in Java security manager didn't prove very useful for the types of security requirements desired for JOLTS. There are several reasons why the Java security manager couldn't be used:

1. The Java security manager can vary on each worker node, which is undesirable because it would require the user to know the security access needed for their program, and the scheduler would need to be much more complicated to handle these types of job requirements.

2. The Java security manager works at too high a level and doesn't allow specific components to be deemed off limits. For example, restricting access to `System.in` while allowing access to `System.out` isn't possible.

3. The Java security manager is runtime based. If a job is already running, it is too late to inform users of security violations. Security checks need to be done when a job is submitted, and *before* it begins execution.

This all leads to the creation of a custom security manager that is configurable by an administrator.

The first level of security is to identify where a computer is located. Only computers that are part of the University of Saskatchewan domain are allowed to submit jobs to the

grid system (in the current JOLTS setup). Thus, it is assumed that users coming from a university machine have already been authenticated by the login system at the university. While only machines at the university can submit jobs, any computer can be a worker node in the grid system.

When a program is submitted to the grid system, either as a jar file or a class file, it is checked to make sure it does not access any forbidden components in the Java library. Refer to Appendix A.1 as to how this is done. Certain components inside the Java library have been designated as forbidden either because:

1. using them can potentially break the submitted program; or

2. using them can potentially break the worker node.

An example of case 1 is class `FileInputStream`. As discussed in Section 3.3, files can cause problems in the grid system; therefore, the special class `GridInputStream` class should be used instead. As a simple example of case 2, user programs are not allowed to call `System.exit` because it would shut down the worker node. A more interesting example of case 2 is the class `ThreadDeath`. When a worker node's load becomes too high, jobs will slowly be terminated until the load falls below some threshold. When a thread is stopped, an instance of error `ThreadDeath` is thrown; consequently, if the user's program was able to catch this instance, it could prevent the job from being stopped prematurely, maintaining the high load on the worker node.

As already mentioned, the security manager is configurable by an administrator. A configuration file contains a list of restricted components. An administrator can change the default values to change how tight or loose security is on the system. Security checks are only done on the server, *not* the worker nodes, so that it can be guaranteed that all worker nodes connected to a server have the same set of restricted components. It was suggested that user groups be created with different security levels; e.g., only users in the highest security level could open sockets. However, this was not done because, as shown in Section 2.1.1.1, authentication can become a tricky issue. Similar functionality can be achieved by having a worker node connect to more than one server, where each server has different security restrictions. Jobs assigned to that worker node will have the security restrictions corresponding to the server that the job was submitted to.

After a job has successfully been checked, the system returns a unique job ID. This ID is used by the user to later retrieve the associated results from JOLTS. This primitive form of security helps prevent someone from getting unauthorized results.

While some effort has been used to prevent malicious code from disrupting the worker node that it is executing on, nothing has been done to prevent malicious code from interfering with a remote computer. This decision was based on the fact that it is impossible to determine the nature of a malicious attack or not. For example, a small web client could be written to be executed on multiple nodes in a grid. Each node would then contact the same web server and request several pages. This setup could be used to easily test the performance of a new web server someone was deploying, a perfectly legitimate use of a grid system. However, if the administrator of the targeted web server was unaware of this coming increase in traffic, it could easily be used as a distributed denial-of-service (DDoS) attack. The point is that by only looking at the source or bytecode, it is impossible to tell which scenario a program is intended for.

3.5 Performance

Most of the performance enhancements in JOLTS deal with trying to make efficient use of the network. The largest concern of these enhancements is with the central server, since it could potentially be communicating with thousands of worker nodes. A secondary concern is the efficient use of threads inside both the server and worker nodes.

3.5.1 Network

The very first choice is to determine whether communication with the server is socket-based or RMI-based. Because there exists the possibility that non-Java programs may wish to submit jobs to the grid, it was decided that connections between the server and clients would be socket-based. The connection between the server and worker nodes could be RMI-based, but the problem is that when a new RMI connection arrives, a new thread might be created (depending on the JVM implementation). Thus, to simplify the design and have full thread control, communication between the server and worker nodes is also socket-based.

With regular sockets selected for communication, the next concern is the protocol

being used. The main consideration is either byte-based or XML-based. XML has the advantage of being much easier to read, debug, and extend, but has the disadvantage that it is verbose and consumes more bandwidth. Byte-based communication is the complete opposite, being very difficult to read and modify; however, if designed properly, it consumes less bandwidth. Byte-based communication was selected because keeping bandwidth consumption to a minimum was deemed more important than making the protocol readable and easily extendable.

With all the socket connections on the server to the various worker nodes, an important question is whether the sockets should be left open or closed after each communication. (This is not a concern between the server and clients because clients disconnect after sending new data to the server.) Because creating new sockets can be an expensive operation, when a worker node first connects to the central server, its corresponding socket connection is left open for the life of the worker node. All data sent to the server from the worker node; i.e., when the worker node initiates the communication, is sent through this socket. Thus, the server will need to continually watch a large number of sockets simultaneously for incoming data. There are three basic ways to deal with this situation:

1. Spawn a new thread for each socket, which then takes the appropriate action when incoming data is detected.

2. Store all the sockets connections in a list, and have a single thread going through a loop checking each socket, to see if it has any incoming data.

3. Use a multiplexor to watch all the sockets. This can be achieved using the *Reactor* pattern described in [21].

Option 1 is unacceptable because creating threads is an expensive operation, and the sheer number of threads required would result in continual thread swapping in the JVM, preventing any of them from making any real progress. Option 2 is a much better solution, but it is basically an active wait, which can consume unnecessary resources. The only remaining option is option 3.

From a conceptual level, a multiplexor operates in a similar fashion to option 2, but an important difference is that there is no active wait involved. Multiplexors usually rely on low-level system calls to watch for incoming data, so they are more common in C/C++.

However, multiplexors were introduced in Java 1.4 with the inclusion of the `Selector` class in the `java.nio` package [12]. For more details on an example using a multiplexor in JOLTS, refer to Appendix A.3.1. For the discussion here, it is only important to know that before a socket can be monitored by a multiplexor, it must be placed in non-blocking I/O mode.

When the sockets are placed in non-blocking mode, any read or write method calls on the socket return instantly. The protocols designed for JOLTS took this into account. Most messages are small enough to reside inside a single TCP/IP packet, so when any data arrives, it is guaranteed to all be there. The one exception to this rule is when sending files. In such a case, reads and writes need to be placed into a loop until all the data has been sent/received correctly. Depending on the reason for the incoming data, it is either handled by the thread executing the multiplexor, or it is executed from inside a thread pool.

3.5.2 Thread Pools

Both the central server and worker nodes need to deal with many concurrent activities. This implies either using multiple processes or threads. Since data is continually being passed between these activities, it is much simpler to use threads instead of processes. Creating a new thread for each activity can be very expensive, so a thread pool is used instead. Thread pools in JOLTS behave as repositories for multiple threads that are available for executing any required tasks. For a task to be executable from inside a thread pool, it needs to implement the `PoolJob` interface.

When a job is placed inside a thread pool, a free thread is selected to execute the job. If no free threads are available, the job is placed inside a FIFO list for processing, once a thread becomes available. When a thread finishes executing a job, the reference to the executed job is lost (eventually garbage collected), but the thread itself does not finish. The thread goes back into the pool as a free thread and either executes a pending job request, or enters a passive wait until a new job is assigned to it.

Multiple thread pools exist in both the server and the worker nodes. The reason for multiple pools on both the server and worker nodes is to allow specific functions to continue executing, instead of being stalled behind a few large jobs that would consume all the threads in a single pool for an extended period of time.

3.5.3 Caching

There are three different types of caching that take place inside the grid system. Two of them are designed to save unnecessary network communication (both done on the worker node), while the third is used to reduce disk space on the server.

The first type of caching on the worker nodes revolves around the checkpoints generated by the client programs being executed. Depending on the size of the checkpoints, it can potentially take a long time to send the checkpoints to the server. In the case when the program can generate new checkpoint files faster than they can be sent over the network (assuming checkpoints are generated in non-blocking mode), consecutive checkpoints can be in the same thread pool awaiting execution. If this happens, the newer checkpoint file takes priority and the older checkpoint is destroyed. This results in a saving of network bandwidth where only the most recent checkpoint is sent to the server.

The second type of caching on the worker node revolves around the `GridInputStream` class. As mentioned in Section 3.3, this class is used for reading files by programs submitted to the grid. When data is to be read from this file, it is first downloaded from the server and cached on the worker node. This cached file can then be used by any instances of the same program that later migrate to this worker node. When the `GridInputStream` object is deserialized with the migrated program, it checks for this cached file and uses it. Once all the cached data is used, more data is downloaded from the server and attached to the previous cached data and the whole process repeats itself.

The final type of caching exists on the server and is designed to prevent excessive disk usage. When a checkpoint file arrives from a worker node, it is downloaded and saved to disk. Because checkpoints could arrive from different worker nodes at different points in the submitted program, the checkpoints can be from different points as well. Checkpoints can potentially be very large, so a check is made for all the stored checkpoint files, and only the most advanced checkpoint is kept.

3.6 Scheduling

Scheduling in JOLTS is not very advanced, because scheduling in grid systems is its own very large research area (*see* Section 2.1.3), and is beyond the scope of this thesis. Scheduling, or the selection of what grid node is to host a job is accomplished inside a

single class, `Scheduler`. The scheduler operates on a simple first-come first-serve basis. When a new job requests to be assigned to a node, the scheduler scans through a list of worker nodes and returns the first one that is willing to host a job. Determining if a node is willing to host a job depends of the following three properties:

1. Whether or not the worker node is currently hosting/executing the maximum number of jobs it is willing to accept — this value is specified by the administrator of the worker node.

2. Whether or not the load on the worker node is below some threshold — this threshold is also specified by the administrator of the worker node.

3. If the node is still alive — this is determined by checking the time since the last heartbeat message from the worker node. If the time is beyond a multiple of the heartbeat time, the worker node is removed from the master list of worker nodes.

When a request for an available worker node is made, it is possible that none of the grid nodes are willing to accept additional jobs. In this case, the request is suspended for the duration of one heartbeat. When the time has elapsed, all the worker nodes will have updated the server with their current load and the process of checking for an available worker node begins again.

While this scheduling algorithm isn't advanced, JOLTS is designed such that different scheduling algorithms can easily be incorporated into the central server by extending the `Scheduler` class and giving a new implementation of the `getAvailable()` method. For example, one addition to the scheduler that could be made is to bias the scheduler into choosing worker nodes that have already hosted a particular job. This approach would help increase the benefit of file caching in the grid. The substitution of one scheduler for another will require the system to be shut down. Scheduler substitution can't take place while the system is running.

3.7 Communication/Coordination

As already discussed in Section 3.2.3, communication among pieces of the same job are done using an object space, defined according to Objective Linda. One of the most common

models used with tuple spaces is the master/slave model, although it is capable of much more. Using this model it is possible to emulate other programming models such a message passing; however, the reverse is not true.

The object space implementation is supplied by the JOLTS system. The programmer only needs to define what objects can be stored in the object space, and what objects can be turned into active objects. Both of these steps are done by implementing the proper interfaces supplied as part of the JOLTS API.

3.8 Name Spaces

Name spaces are a familiar topic to C/C++ programmers, but are rarely encountered by Java programmers. However, in grid systems name spaces become relevant even to Java programmers. Name spaces deal with how to group together associated classes, without interfering with other groups of classes that can potentially share the same name. In normal Java programs, when a class is loaded, the loader first checks to see if it is a system class, defined somewhere inside the `java.lang` package. This is done by moving up the class loader hierarchy. If it isn't found, it then repeats the process checking the caches of the loader in the same hierarchy until either a cached copy is found, or one of the loaders knows how to load the class (a `ClassNotFoundException` is thrown if no loader could load the class). A problem can arise when different implementations of a class with the same name are loaded.

Consider a scenario with two programmers, Alice and Bob, who each want to submit a program they wrote to the grid for execution. Both of them named their program the same, say `Foo`. Alice submits her program first, which passes through the security checks and is assigned to node A. When A gets Alice's program from the central server, it loads it and begins execution. Bob then submits his program, which also passes the security checks and gets assigned to the same node as Alice's program. Now when A goes to load Bob's program, as it moves up the class loader hierarchy, some object will already know how to load a class called `Foo`, so it loads it and the program starts executing. The problem is that the cached version is Alice's `Foo`, and not Bob's. Thus, when Bob goes to obtain his results from the grid, he will get results based on a run of Alice's program, not his own. If checkpointing is used in either version of `Foo`, things can become even more complicated.

The solution to this problem is a customized class loader.

JOLTS actually has two customized class loaders: one deals with loading a single class file, and the other is responsible for loading entire jar files. Both class loaders are also responsible for helping implement the malicious code protection discussed in Section 3.4. Here is how the previous example will be handled with the customized loaders. When Alice's program is started on worker node A, an instance of one of the grid class loaders is created. This loader is then responsible for loading Alice's program, based on the ID of Alice's job. Any other classes that need to be loaded during the execution of Alice's program will also use this class loader. When Bob's program arrives, another instance of one of the grid class loaders is created, which then loads Bob's program based on the ID assigned to his grid job. When either Alice or Bob go to obtain their results from the grid, they will both get results generated by their own instance of `Foo`.

Thus, the ID assigned to a grid job also helps act as part of a name space, making each program unique even if they have classes with the same name. The use of a customized loader, along with the job IDs makes name spaces really a non-issue in JOLTS. The main point to remember is that programmers are totally unaware that this problem even exists.

Chapter 4

JOLTS Detailed Design

While the previous chapter gave a brief listing of the features of the JOLTS system, this chapter is meant to give a high-level view of the design of the system. Low-level implementation details are kept to a minimum in this chapter. Some of the more interesting implementation details, however, are given in Appendix A including large code examples. Conceptually the JOLTS system, consists of three main components: a client, a server, and one (preferably more) worker nodes. First, it is important to clarify some of the terms that will be used throughout this chapter.

- Worker node — the computer that executes the user's job(s), or sub-jobs in the case of a multi-part job.
- Server — a large computer that is responsible for accepting jobs from clients, returning job results, and sending out jobs/sub-jobs to the various worker nodes.
- Client — the *computer* that submits jobs to the server.
- User — the *human* that uses the client to submit jobs.
- Programmer — the human that is responsible for writing the programs that are turned into jobs and submitted to the JOLTS system. The user and programmer are not necessarily the same person, but they can be.
- Job — programs that are submitted by a user. There are four main types of jobs, in increasing complexity:
  1. Simple/Sequential
  2. MISD

3. SIMD

4. Object Space

All but the first consist of many smaller sub-jobs that make up the larger job. Sub-jobs that are part of a larger object-space job are referred to as *active objects*, as dictated by Objective Linda.

The JOLTS system from the user's (and programmer's) point of view is rather simple. The programmer writes the program(s) using the supplied JOLTS API. The user then uses the client to submit the job to the server using either a command-line or graphical interface. The server sends back either a job ID that is used when retrieving the results for the job, or an explanation of why the job was rejected. The user can have the client either display the results of job, or the results can be incorporated into another program directly, if desired (*see* Section 4.2). Behind the scenes, hidden from the user, many operations are performed to execute the submitted job.

The server does several types of verification on the submitted job (and the client who submitted it), before sending the job to worker(s) for execution. Once the verification is complete, sub-jobs, or the entire job in the case of sequential jobs, are sent to various worker nodes for execution. As the results are returned by the worker nodes, the results are saved on the server and more sub-jobs are sent out until the entire job is complete. Client requests for results can then be returned with the actual results, instead of a message telling them to try again later.

While this "behind the scenes" view may also appear rather simple, in reality is requires complex interactions between many classes. The JOLTS system consists of 108 Java classes, and nearly 11,000 lines of code. The remainder of this chapter is meant to give a high-level view of the various modules that make up the JOLTS system, and the design decisions that went into their creation. For a detailed look at some of the more challenging/interesting *implementation* problems refer to Appendix A. A layered view of the JOLTS system is given in Figure 4.1. The system is composed of four main modules:

1. Client side

2. Worker side

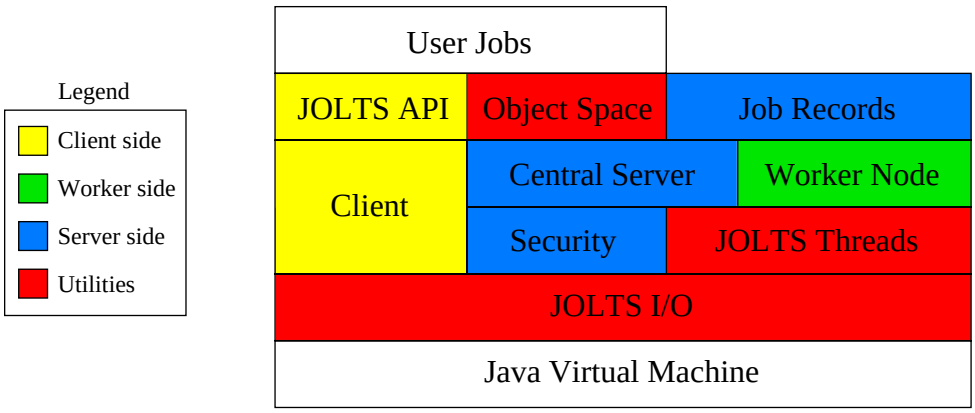

Figure 4.1: A layered view of the JOLTS system architecture

3. Server side

4. Utilities

All but the first of these modules is further decomposed into sub-modules. Coupling between the various modules has been kept to a minimum; however, it is neither possible nor desirable to eliminate coupling entirely. Depending on which component (client, server, or worker) is being used, only the corresponding module (as well as the utilities module) is needed to execute that component. Each of the four modules will now be discussed in turn.

4.1 JOLTS Utilities

This module, and its sub-modules, contain many of the classes and interfaces that are required for the client, server, and worker to function properly. Figure 4.2 shows the classes at the top level of this module, along with some of the relationships between them and classes outside this module. Two important components at this level are the `GridLoader` hierarchy and the `CheckpointMech` interface.

The JOLTS system is capable of running several independent jobs from distinct users at the same time. Part of a job submitted by a user is the binary for the job as either a class file or a jar file. Regardless of the type of file submitted, the Java Virtual Machine (JVM) will be unable to see the supplied binary file because it isn't in any of the default paths the JVM looks in, when it attempts to load a class. Thus, a custom class loader is required for loading the user's binary file into the JVM. As seen in the hierarchy of

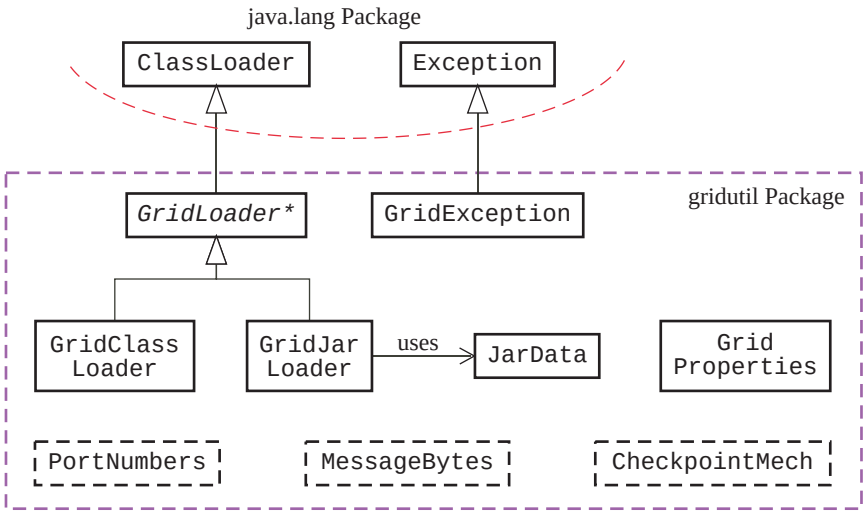

Figure 4.2: The top level of the utilities module

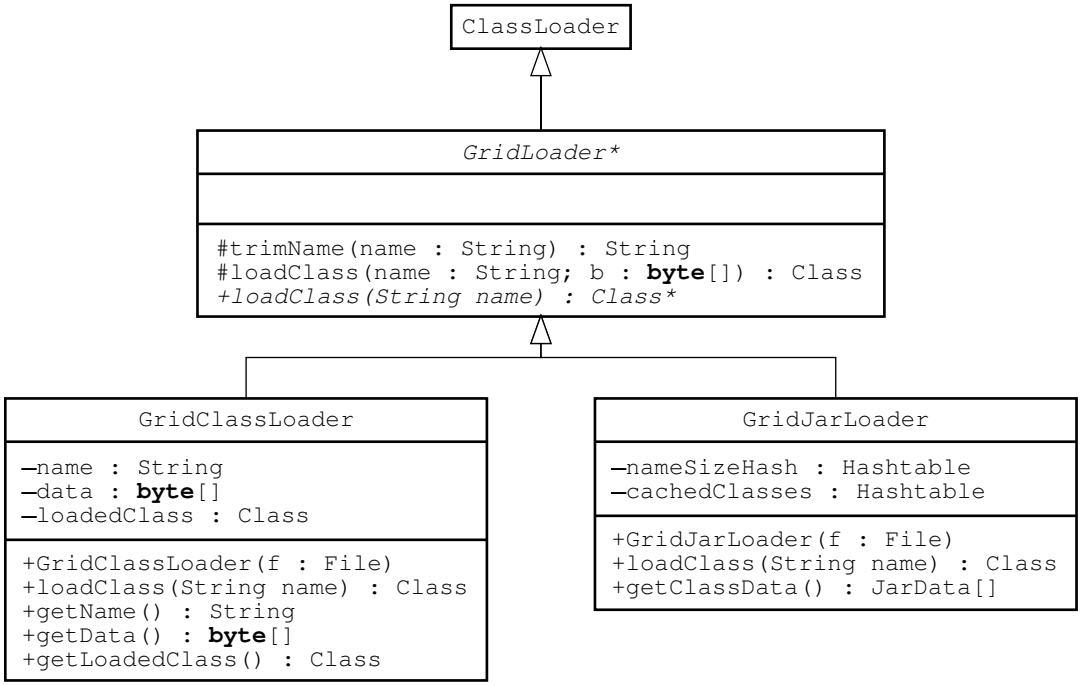

Figure 4.3: The `GridLoader` hierarchy

Figure 4.3, custom loaders exist for both jobs submitted by class files and jar files. Each job submitted to the JOLTS system will be loaded by a new instance of the appropriate class.

An additional benefit of using a custom class loader for each job is it eliminates the possibility of naming conflicts between distinct jobs that happen to be run on the same worker node. Solving this problem is highly language-dependent; e.g., C# uses name

spaces (*see* Section 3.8). When the JVM attempts to load a new class, it is the responsibility of the class loader to find the requested class description and load it into memory. When multiple class loaders exist (as is the case in JOLTS), knowing which class loader will service the request is very important. First the class that originated the request is found. The loader for the class originating the request is used to service the request. If this loader is unable to service the request, the request is passed back up the runtime loader hierarchy for servicing. If no loader can find the request class description, an error is thrown by the JVM.

Consider the case where two programs, `A` and `B` are each running on the same worker node. Each program has been loaded by its own class loader; e.g., `loaderA` and `loaderB`, respectively. Suppose program `A` needs to create a new instance of class `B`. Because `A` made the request, `loaderA` is responsible for loading the class `B` instance. The existing class `B` instance, loaded by `loaderB`, is *not* used because `loaderA` and `loaderB` are completely disjoint (and thus, so are the programs for which they are responsible). In the situation where multiple sub-jobs from the same multi-part job are on the same worker node, they will all be using the same class loader. These custom class loaders will play an important role when dealing with object spaces.

As mentioned as the start of this section, the other important piece in the top level of the utilities module is the `CheckpointMech` interface. To create a checkpoint, the user's job must pass a number to the system to indicate the version of the checkpoint. This was done for several reasons:

- It saves disk space when storing checkpoints on the server, only the most recent is stored.
- It allows bandwidth to be saved when multiple checkpoints for the same job need to be sent over the network at the same time, only the most recent is sent. Consecutive waiting checkpoints can only occur if the user has elected to create non-blocking checkpoints. This reason will be discussed shortly.

Checkpoints are an extremely useful feature, but very few systems support them. To prevent programmers from abusing the checkpoint mechanism, the decision was made to make the version numbers for the checkpoints only 16-bits long, instead of the more common 32-bit numbers. When the programmer is writing a program that uses checkpoints

they will most likely have to typecast the checkpoint version number passed to the system. The *intention* is that the programmer will realize there are a limited number of checkpoints they can create when doing the typecast, and the programmer will spend some time determining when a checkpoint *really* needs to be created. For example, if a program is manipulating a lot of records, instead of creating a checkpoint after every record is modified, the programmer will hopefully space out the checkpoints; e.g., one after every 15000 records have been modified.

Depending on the nature of the user's program, it is possible for the program to generate checkpoint data faster than the data can be sent over the network and stored on the server. If this happens a queue will form on the worker node for all the checkpoints that need to be sent to the server. This queue is in fact a priority queue, where newer checkpoints take priority over older checkpoints in the queue. The newer checkpoint removes the older checkpoint entirely from the queue. Multiple checkpoints can only occur in the queue if the user has elected to create non-blocking checkpoints.

Sending checkpoints to the server is an I/O bound operation, while the user's job is usually CPU bound. By default, checkpoints are sent to the server using a separate thread. This allows the user's job to continue executing while the checkpoint is being sent. However, there are times when it may be desirable for the job to know that a specific checkpoint has reached the server before the job continues executing; e.g., in object space jobs. In such a case the checkpoint mechanism can be placed in blocking mode using `setBlockingMode(true)`. The blocking mode of the checkpoint mechanism can be obtained using the function `inBlockingMode()`. When the checkpoint mechanism is in blocking mode, any request made for a checkpoint will only return once the checkpoint has successfully been transmitted to the server.

4.1.1 Task Utilities

Possibly the most important utilities sub-module, even more so than the object space (*see* Section 4.1.4) is the tasks sub-module. The classes that makeup this sub-module, and their relationships, are given in Figure 4.4. A *task* is a unit of operation that needs to be performed, ranging from executing a user's job to the heart beater on a worker node. Most tasks are independent of each other, and thus can be run in parallel. The pieces of interest in this module are the interface `PoolJob` and class `GridThreadPool`. Many of the classes

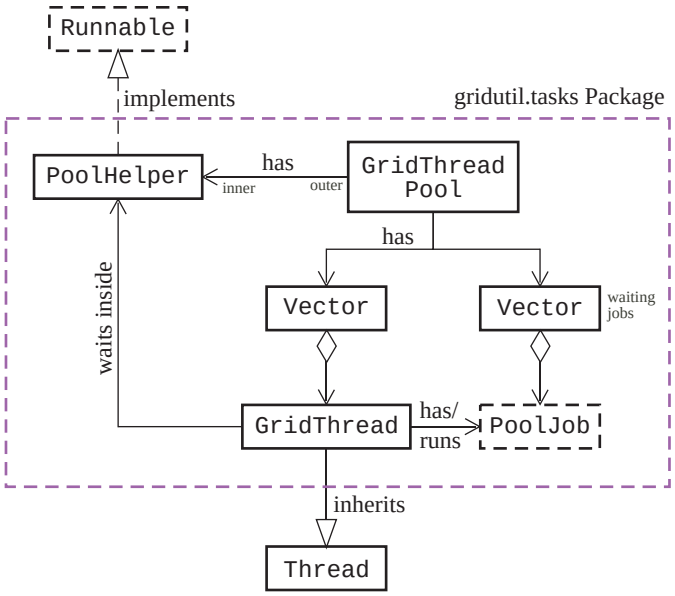

Figure 4.4: The classes of the tasks utilities sub-module

scattered throughout the JOLTS system are descendants of `PoolJob`, which allows these descendants to be run in parallel from inside a thread pool. The `GridThreadPool` class, along with the other classes/interfaces in this package create a concrete implementation of the Thread Pool design pattern. The pool is initially created with several threads inside that can execute any descendants of `PoolJob` placed inside the thread pool.

```
GridThreadPool incomingJobsPool = new GridThreadPool("incoming job requests",
        5, Thread.NORM_PRIORITY);
⋮
PoolJob task = new IncomingCodeJob(sockChannel);
incomingJobsPool.addNewJob(task);
```

If there are no free threads, the tasks just wait in a queue until a thread inside the pool becomes free. When a task finishes, the executing thread doesn't disappear, it stays inside the pool waiting for another task; if no tasks are waiting the thread is put to sleep until a tasks arrives. Thus, the expensive operation of creating threads is isolated to the initialization of the pool, and not scattered through the life of the system each time a new task needs execution. The `GridThreadPool`'s services are used extensively throughout the server, and to a lesser extent in the worker nodes and is a key component in the Reactor Pattern.

4.1.2 Constant Pool Utilities

The classes in this sub-module are used exclusively by the security manager on the server, and are used in the creation of the constant pool stored inside an individual class file. The security manager uses this constant pool information to determine if a class accesses any restricted components in the Java library (*see* Section 3.4). For details on how this module works, refer to Appendix A.1.

4.1.3 I/O Utilities

Like the tasks sub-module discussed in Section 4.1.1, the I/O sub-package is used basically everywhere, as indicated by its placement in Figure 4.1. The most important class in this sub-package is `GridFileStorage`. This class is a singleton, only one instance exists on each computer of the JOLTS system. Because the various JOLTS components can run anywhere the JVM has been ported, the `GridFileStorage` class is designed to abstract away file access to the underlying hardware. This is especially important for storing various temporary files in the properly designated "temp" directory, whose contents are needed both locally and remotely. Thus, a `GridFileStorage` object also has the capability to send and receive files over a socket channel.

Another important class in this sub-module is `GridObjectInputStream`. Its job is very simple, but also very important. Since JOLTS operates on programs that are actually objects, these objects need to be passed around the system at various points. When objects are deserialized from an input stream, the default system loader is used to deserialize them. As already mentioned in Section 4.1, the default system loader can't be used to deserialize jobs. To get the JVM to use a different class loader to deserialize objects, a custom object input stream is needed; thus, why class `GridObjectInputStream` exists. It reads in bytes from the stream and deserializes the incoming object using the proper custom loader.

The final relevant class in this sub-module is `GridInputStream`, already discussed in Section 3.5.3.

4.1.4 Object Space Utilities

The relationships between the classes and interfaces in this sub-package are given in Figure 4.5. All of the components in this package (with the exception of class `ObjectSpace-`

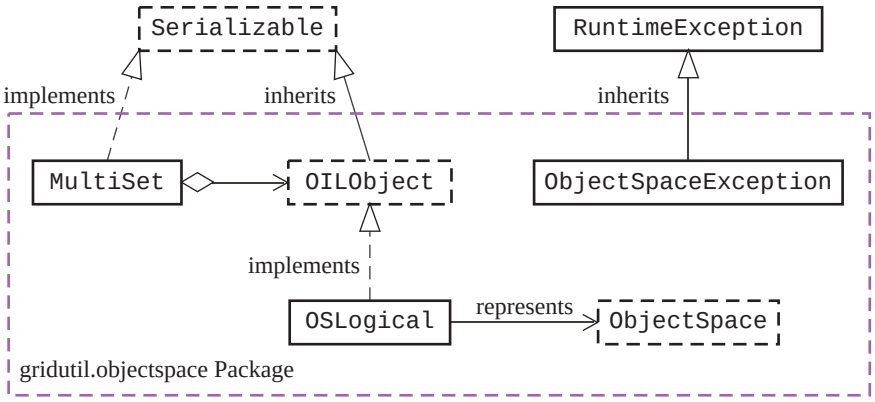

Figure 4.5: The classes and interfaces of the object space utilities sub-module

`Exception`) are defined according to the Objective Linda specification discussed in Section 2.2.5. Each of these components will now be discussed in turn.

The `OILObject` interface defines the single `match()` method. It is defined as an interface so that programmers that wish to use the object space can still extend a class, if they need to. Only descendants of `OILObject` can be placed in the object space. It is important to note that the `ObjectSpace` interface is *not* a child of `OILObject`, because object spaces can't be nested inside of each other. To overcome this limitation, the `OSLogical` class can be used.

An `OSLogical` instance is used to take the place of an object space inside another object space. This creates the *illusion* of nested object spaces. An active object creates an `OSLogical` object by giving it a String identifier and the object space it is supposed to represent. The newly created `OSLogical` object can then be placed in any available object space. Any active object can then retrieve this object and perform an `attach()` operation to get a reference to the object space the `OSLogical` object represents. In this manner active objects are able to gain access to object spaces besides their default `self` and `context`.

The `ObjectSpace` component is defined as a interface because both the workers and server have different implementations, described in detail in Appendices A.2 and A.3, respectively.

4.1.5 Additions to Objective Linda

Two additional features have been added to the object space that aren't part of the original Objective Linda specification:

```
public int rm(OILObject template, int min, int max, long timeout)
public int rmOut(OILObject template, int min, int max, long timeout, MultiSet m)
```

The first method, `rm()`, is very similar to the default `in()` operation, except that instead of returning a multiset like `in()`, `rm()` returns the *size* of the multiset, which indicates the number of objects removed from the object space. Originally, objects could only be removed from the object space using `in()`, which returns the removed objects in a multiset, regardless of whether the objects were needed or not. This is a waste of network bandwidth if the multiset's contents aren't needed. Thus, the `rm()` method should be used instead, if the returned objects aren't needed.

As just mentioned, the `in()` operation is used to remove objects from the object space. Often an object is removed from the object space and a duplicate object in a different state is immediately placed back into the object space using the `out()` operation. Because these are two separate operations, it can potentially cause timing problems with other active objects that are looking for either the objects just removed or for the new objects being inserted. The `rmOut()` method listed above is meant to deal with this problem. It performs a `rm()` operation followed by an `out()` operation, that are executed as an atomic operation on the object space. The result returned by method `rmOut()` is the number of objects removed by the `rm()` half of the operation. If the `rm()` half fails; e.g., the minimum number of matches were not found, the `out()` half will *not* be executed. If the `rm()` half works, the `out()` half is guaranteed to succeed. The `rmOut()` operation is beneficial when creating the illusion of changing the state of object(s) already in the object space, which isn't normally possible as a single operation.

The Objective Linda specification doesn't state the behavior of the features for the multiset, so a best attempt was made to make class `MultiSet` compatible. The public features provided by the JOLTS `MultiSet` are:

```
public OILObject get(int i)
public OILObject get(OILObject obj)
public void put(OILObject obj)
public int numItems()
public boolean equals()
public String toString()
```

There is no remove method for the JOLTS multiset. Once method `put()` places an object

into a multiset, it stays there permanently. Some versions of Objective Linda have a `get()` method with no arguments that returns *and* removes the first item in the set. This feature was deemed undesirable, because it is a function with side effects. Thus, this is why there are multiple `get()` methods listed above.

The final class in the object space sub-module is `ObjectSpaceException`. This exception is only thrown when a precondition is violated on an object space. The main preconditions are:

1. All objects passed to any object space method must be non-null.

2. For the methods requiring a timeout value, $0 \leq timeout \leq infinite\_timeout$ must hold true.

3. For the methods performing matches on a template, $0 \leq min \leq max \leq infinite\_matches$ must hold true.

4. Each item in the multiset passed to method `eval()` *must* be a descendant of interface `ObjectSpaceJob`.

An `ObjectSpaceException` instance is a runtime exception, so programmers don't have to deal with handling this exception, if they don't want to. The current active object will crash, if it isn't handled, but all the other active objects will remain unaffected.

4.2 Client Side Module

This module contains both the classes used to submit/retrieve jobs from the server (both CLI and GUI versions), as well as the main JOLTS API needed by the programmer when creating programs to run on the JOLTS system. Figure 4.6 shows the main classes and interfaces available to the programmer. Depending on the nature of the program being written, I/O or object space classes may also be needed (*see* Sections 4.1.3 and 4.1.4, respectively). Most of the components in this module are interfaces. A few abstract classes also exist. After building several test programs, a pattern began to emerge in the implementation of the test programs. These common methods were used to create the abstract "template" classes to save the programmer from redundant implementations.

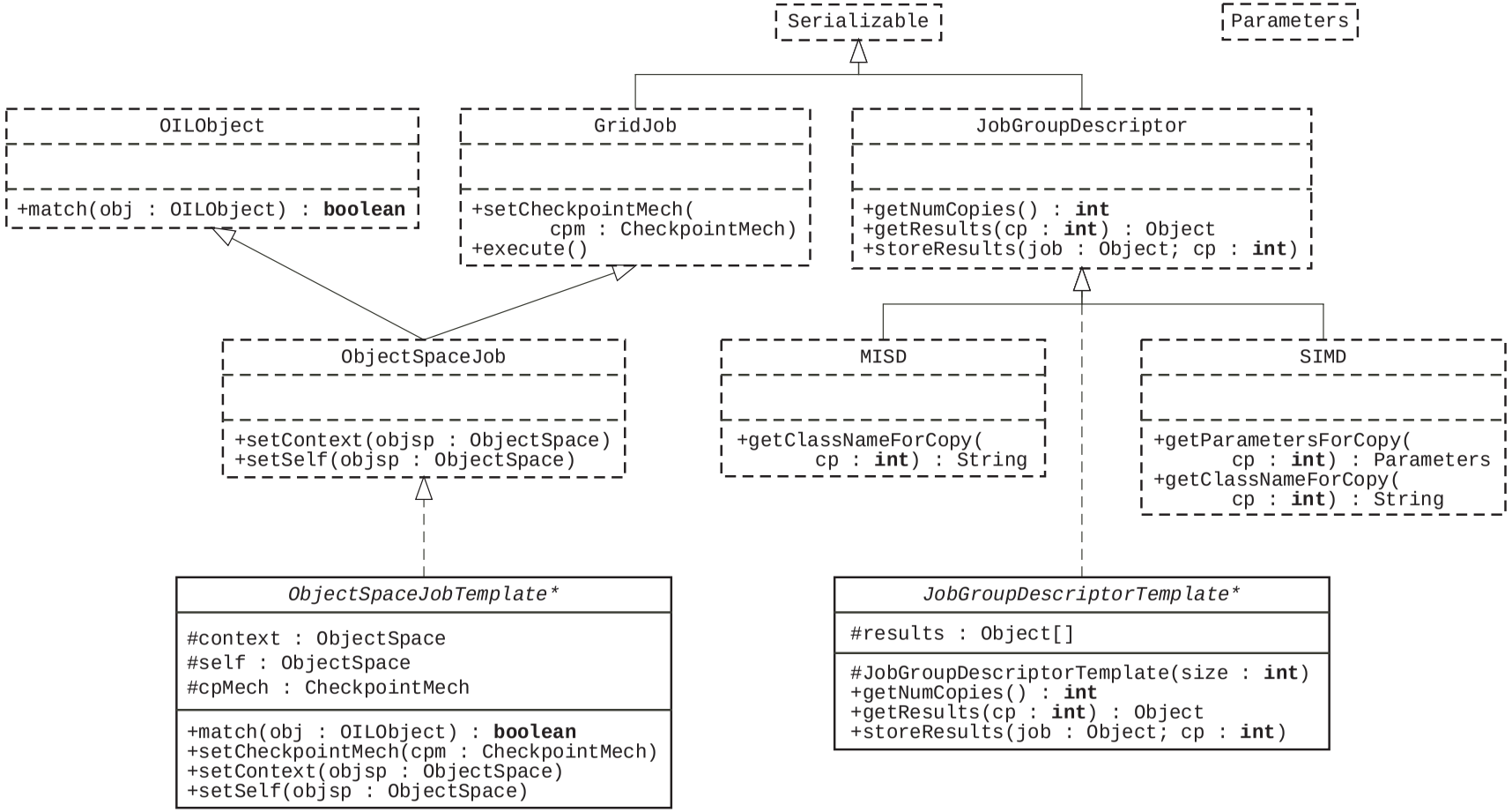

Figure 4.6: The main JOLTS API available to programmers

It is important to note that while the client is capable of submitting, retrieving, and displaying jobs, it isn't limited to working in isolation. If a program created by a programmer needs the capabilities offered by JOLTS for only *part* of its execution, it is possible to incorporate the class `GridClient` into this external program to interact with the server on behalf of the user's main program. For example:

```
GridClient client = new GridClient("MyProg.jar");
int jobID = client.submitJob("StartClassName", inputFiles);
... // perform other operations
Object obj = client.retrieveJob(jobID);
if(obj instanceof ... ) // make sure that results are actually returned
{
    ...
}
```

As mentioned earlier (page 57), the JOLTS system supports four main job types. Each job has a corresponding class/interface in this module that the programmer needs to inherit from to get his/her program to run on the JOLTS system. The name of the child class goes on the right-hand side of the **`instanceof`** in the above code fragment.

Most grid systems that support executing parameter experiments, such as Condor, require special configuration files be submitted with the job. These configuration files are used to describe information about the job, such as the minimum amount of memory and processor type required for the job. While these type of configuration files are beneficial, they are often confusing to create for beginners because the user may not have the skills required to gather the information needed in the configuration file. JOLTS requires no such configuration file in the hopes it will lessen the learning curve for programmers to start using the system. Also, these configuration files deal with hardware/software requirements, while the JOLTS system is designed to intentionally abstract that away from the programmer, so configuration files are not necessary.

Simple, or sequential jobs are created by implementing interface `GridJob`. This interface is used for jobs that can't be decomposed into parallel components, *and* for the smaller components of a job that *has* been decomposed into sub-jobs. It defines a method, `execute()`, where the job starts and a method to pass in a checkpoint mechanism (*see* Section 3.3), called `setCheckpointMech()`. There are two reasons why there is a `set` method for the checkpoint mechanism instead of passing it to a constructor.

1. It is very difficult to enforce constructor parameters in Java. While it *can* be done, allowing any additional parameters to the same constructor makes it even more

difficult. Also, if there are multiple constructors that have the proper parameter type, it isn't always clear which one should be used to create the object.

2. If a job is resumed from a checkpoint, a new checkpoint mechanism needs to be given to the object, since the checkpoint mechanism *isn't* checkpointed along with the rest of the job (*see* Appendix A.4 as to why). Since the object already exists, using the constructor to pass in the new checkpoint mechanism isn't feasible. Thus, the only remaining option is to use a `set` method. **Note:** Whether or not the object stores the argument passed to the "set" method is up to the programmer. If they don't want to use the checkpoint feature, the argument can be ignored.

The `ObjectSpaceJob` interface adds methods for setting both the "self" and "context" object spaces. The reason for these two "set" methods is similar to that just given for the `setCheckpointMech()` method. When a *regular* object is to be converted to an *active* object by way of the `eval()` method on an object space, it needs to be given access to the proper object spaces. Again, since the object already exists, using the constructor is not an option. Additionally, only *active* objects should be assigned a "self" object space, and the object only turns into an active object when it is passed to `eval()`. If every object that was created instantly had its own object space, it would end up wasting a lot of memory on the server where the object spaces actually exist. Because active objects rarely need to match each other, the abstract class `ObjectSpaceJobTemplate` has an empty implementation of `match()` that always returns false. It also has methods and fields for setting and storing the default object spaces.

Most grid system don't have a way to allow communication between jobs, but they do support independent sub-jobs. The JOLTS system supports independent sub-jobs, specifically parameter experiments, using the `MISD` and `SIMD` interfaces. Because JOLTS already supports simple jobs using the `GridJob` interface, the `MISD` and `SIMD` interfaces merely acts as a way to describe the sub-jobs (each of which is a `GridJob`), and a place to assemble all the completed results. A `MISD` job adds very little new functionality over the simple jobs. It is basically a convenient way to submit multiple simple jobs at once. A `SIMD` job, on the other hand, adds some important features.

All of the `SIMD`'s sub-jobs are started using a constructor that takes a single `Parameter` argument, while regular jobs and `MISD` sub-jobs are started using a constructor that

takes no arguments. This parameter is the "data" in the Single Instruction Multiple **Data**. Each sub-job is given a different instance of `Parameter`, determined by the `SIMD` descendant written by the programmer. Storing and retrieving the sub-jobs was often identical between the `MISD` and `SIMD` interfaces during testing, so the abstract class `JobGroupDescriptorTemplate` was created. This class is defined as abstract, even though it has no abstract methods, because its children are meant to be used in conjunction with the `MISD` or `SIMD` interfaces. The child class uses the abstract parent for storing/retrieving the sub-jobs, but it needs the interface parent to inform the system what type of parameter experiment is to be run. Consider the following class header:

```
public class MySIMD extends JobGroupDescriptorTemplate implements SIMD
```

The class `MySIMD` can then focus on describing what the parameters are for the sub-jobs, instead of worrying how to store and retrieve results.

Figure 4.6 shows the storing and retrieving of results is declared to be using class `Object`. Ideally the objects being stored as results would be descendants of the `GridJob` interface, allowing the store and retrieve method signatures to be:

```
public GridJob getResults(int cp)
public void storeResults(GridJob job, int cp)
```

However, using these signatures instead of the ones given in Figure 4.6 has one important flaw: dealing with crashed sub-jobs. If a sub-job crashes (*not* the worker node executing the sub-job) because of an exception; e.g., a null pointer exception, the thrown exception is caught and is returned as the result for the sub-job. Because the interface `GridJob` is not a descendant of class `Exception`, if an exception is stored using the method `storeResults()`, the *entire* job would crash because of a typecasting exception. Using the method signatures from Figure 4.6, it is possible to store anything as the results for a sub-job, including exceptions. The other option that was considered when designing this capability of the system was to have the JOLTS system just leave the results for the crashed sub-jobs as **`null`**. While this would have made the JOLTS system easier to implement, it would prevent the job's programmer from obtaining any useful feedback about a potential problem in a sub-job. The programmer can now use the exception result to help determine why the sub-job crashed.

A final point about the client module before moving onto the worker node is that anything that needs to get sent over the network (or checkpointed) must be serializable. Only

so much can be checked before executing a job. If a job/sub-job is serialized that has non-serializable fields, the results for the job/sub-job will be an exception. A *full* understanding of how serialization works isn't required by a programmer to use the JOLTS system, but the programmer should be aware of potential serialization problems and prevent them from occurring; e.g., by using **`transient`**.

4.3 Worker Node Modules

At a high-level, the responsibility of a worker node is to receive jobs and sub-jobs from the server, download any required files, execute the user's job, and send the results back to the server. Periodically, the worker also must send new load and job statistics to the server, which are used by the scheduler on the server to determine where to send pending jobs.

The JOLTS worker node modules consists of three main areas:

1. Statistics collection
2. Execution of independent jobs
3. Execution of object space jobs

Each of these areas will now be discussed in turn.

4.3.1 Statistics Collection

Sending periodic updates about the load and current jobs on a worker node is done using the Heart Beat pattern, implemented using the `Heartbeater` class. Because the method used to determine the current load of the computer varies between operating systems, this is one of only two locations (the other being class `GridFileStorage` discussed in Section 4.1.3) where the underlying hardware and operating system come into play. Fortunately, it doesn't cause much of a problem, as the OS-dependent calls are hidden away using the Factory pattern in the `WorkerNodeProperties` hierarchy, shown in Figure 4.7. If JOLTS is to be run on a new operating system, all that is needed is a new child of `WorkerNodeProperties`. Users of the hierarchy don't need to know about the child classes, only the parent class `WorkerNodeProperties`. They merely obtain the proper child using

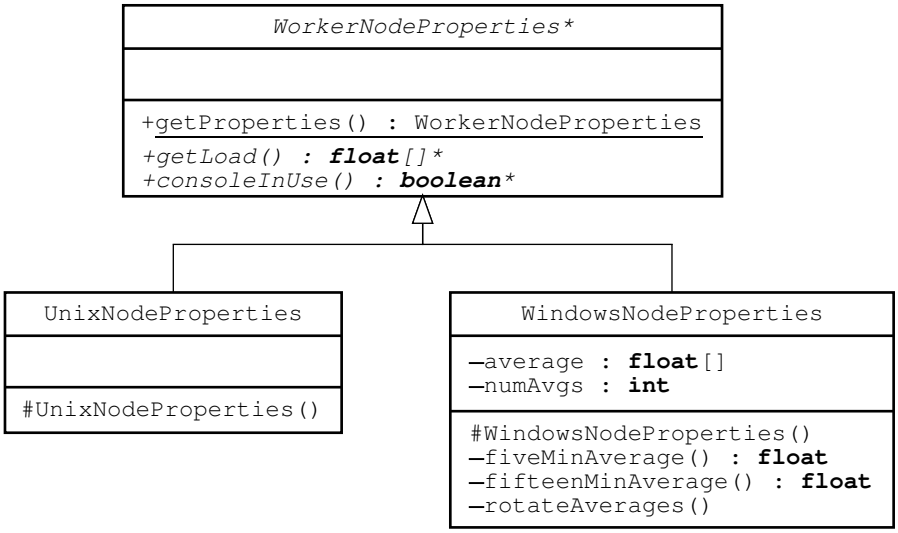

Figure 4.7: The `WorkerNodeProperties` hierarchy

method `getProperties()` defined in the abstract class `WorkerNodeProperties`. Only this class knows which child should be created.

One of the main uses of the heart beater in network systems is to send some type of periodic notification between computers to inform one computer that the second computer is still running. The type of information contained in this heart-beat message depends on the nature of the application.

Because heart-beat information is outgoing and fairly frequent, it is beneficial to leave the socket used for the communication open for the life of the worker node. This also implies the other end of the socket is permanently open on the server to receive the communication. To help prevent opening excessive sockets on the server, heart-beat data, job results, and checkpoints are all sent over the same socket. This also has the advantage of acting as a security measure on the server, since only worker nodes that are already verified can send data to the server. Because many threads can be competing for this one socket on the worker node, all writing to this socket (there is no reading) is done inside a critical section. Without this protection, the server could see scrambled data from many different jobs all arriving at once. Competing with class `HeartBeater` for the outgoing socket is class `ReturnResults`, discussed in Section 4.3.2.

One important feature of class `Heartbeater` is its ability to kill any executing job on the worker node. Because the `Heartbeater` is the only class that knows the load of the worker node, it is the responsibility of this class to indicate that a job must be killed, if the load

has passed its maximum specified threshold (specified by the worker node administrator). This process continues until the current load is under the threshold. The server will detect which job has been killed in this manner and will attempt to restart/resume the job on another worker node. This job-killing feature is the reason why JOLTS worker nodes don't have to be, and in fact *shouldn't* be run at a low priority, i.e., be "niced." If the worker node is run at a low priority, any jobs running on that node will receive fewer cycles when the node is under high load; thus, increasing the time until the job/sub-job is completed. However, if the worker node is running with regular priority and the machine load becomes too high, the worker node will begin killing jobs; in effect causing them to move somewhere else where there are more free cycles available. This can be represented by a simple equation where there are more free computers than jobs to execute:

$$n \text{ jobs} \times \frac{1}{n}\text{CPU} < n \times (1 \text{ job} \times 1 \text{ CPU})$$

Any number of jobs, n, will be faster if they are each run on their own CPU, as opposed to all having to share one CPU.

The migration of executing jobs benefits both the job and the person using the worker node as a desktop computer. The job moves to a worker node with more free cycles, and the cycles released by the job can be applied to whatever the user is doing that caused the load to surpass the specified threshold.

4.3.2 Handling Independent Jobs

The worker node listens to a server channel for incoming (sub)jobs from the server. This channel, and its resulting worker job are done using the Reactor Pattern. While this may seem like overkill for the worker node, this pattern was used to keep a constant appearance with the server, where the Reactor pattern is essential to its operation. Depending on the type of incoming job, the appropriate class of the `ExecuteNewGridJob` hierarchy, shown in Figure 4.8, is used to begin servicing the job. An explanation of each class is as follows:

- `ExecuteNewGridJob` — a class used to execute *new* simple/sequential jobs, and MISD sub-jobs.

- `ExecuteNewSIMDJob` — a class used to execute new SIMD sub-jobs. It is much more complicated that the previous class because it needs to download the parameters

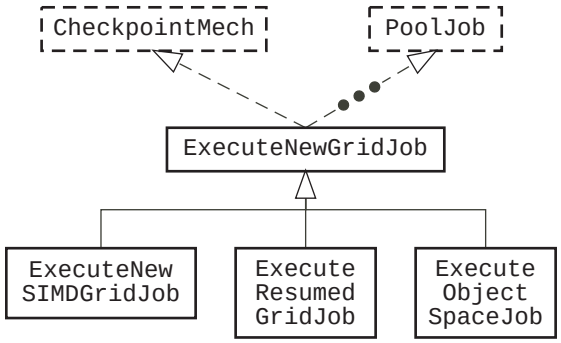

Figure 4.8: The `ExecuteNewGridJob` hierarchy

from the server, and find the proper constructor for the sub-job, before it can create it and begin executing the sub-job.

- `ExecuteObjectSpaceJob`—a class used for executing active objects (*see* Section 4.3.3).
- `ExecteResumedGridJob` — a class used when *resuming* any of the four jobs types from a checkpoint file.

Each of these objects is executed from inside a thread pool (*see* Section 4.1.1). Regardless of the job type, a record is created about the sub-job. The object representing the user's job is obtained (created or downloaded) in conjunction with a custom class loader. The job is then given a checkpoint mechanism, and finally, the user's job is started executing. When either final or checkpoint results need to be sent to the server, an instance of `ReturnResults` is used. Instances of this class are executed from inside a separate thread pool. The reason a separate thread is used is that the `ReturnResults` instance can potentially block while waiting to obtain a lock on the outgoing channel (recall from Section 4.3.1 that it competes with the `Heartbeater` for the outgoing channel). If the `ReturnResults` instance is sending checkpoint data and *wasn't* in a separate thread, the job that wanted to create a checkpoint would also block until its checkpoint was sent to the server, which is a waste of resources. However, when blocking checkpoints are used, the `ReturnResults` instance *doesn't* operate in a separate thread. Once the final results for the job are queued to be sent to the server, the thread that executed the user's job returns to it's thread pool.

The checkpoint mechanism that is passed to the user's job is a reference to the same object that is responsible for executing the user's job, an instance from the `ExecuteNew-`

`GridJob` hierarchy. When a checkpoint is requested by the job, the job is serialized and written to disk with a checkpoint version number becoming part of the file name. A `ReturnResults` object is created to send the checkpoint file to the server, while the job continues executing (assuming non-blocking checkpoints). Because the user's job continues executing while the checkpoint is waiting to be sent, it is possible another checkpoint could be created before the previous one was sent (recall that a lock on the outgoing channel must be obtained before the data can be sent). If this happens, the newer `ReturnResults` object will delete the previous one waiting in the queue for objects trying to obtain a lock on the outgoing channel. This saves bandwidth by only sending the most recent checkpoint to the server. As beneficial as checkpoints are to user jobs, they require a little more caution when used with object space jobs.

4.3.3 Executing Object Space Jobs

As just mentioned, object space jobs need to be more cautious when using checkpoints. The reason is that although the active object is checkpointed, the object space(s) used by the active object are *not* checkpointed. The reason is that it would require checkpointing *all* the active objects and object spaces associated with the entire object space job at the same time to create a snap-shot of the job. This is impractical to do for many reasons. The biggest reason is that it would be impossible to *guarantee* that every active object was in a safe, stable state when the checkpoint was requested.

Starting an object-space job on a worker node is very similar to resuming a checkpointed job and involves the following:

- The active object is downloaded from the server.
- The checkpoint mechanism is set.
- The `self`, and `context` if necessary, object spaces are set.
- The active object is started executing.

Checkpoints are created the same way as non-object space jobs, and when an active object finishes, a message is sent to the server indicating the active object's completion. What makes object space jobs unique is the object space that all the active objects, potentially

scattered across various worker nodes, use to communicate with each other. The actual object space is maintained on the server for several reasons:

1. Stability — it is assumed the server won't crash, unlike the worker nodes.

2. Resources — the server usually has more memory and a larger/faster bus than the worker nodes, allowing it to handle communication with a large number of worker nodes simultaneously.

3. Location — the server is a known location, that can't change during the life of the worker node. Thus, all worker nodes know which computer is the object space server.

The object space on the worker nodes, used by active objects, are really just stubs that communicate with the real object space on the server over socket channels. This network communication is completely abstracted away from the user's job. For details on how this is implemented, refer to Appendix A.2.

An interesting capability of the object space is that of timed *function* execution. Every object space operation has a timeout value associated with it. If the operation can't be performed in the requested time, a default value is returned. These are *function* operations, *not* procedure operations; however, the same techniques can be applied to procedures by turning them into Boolean functions that return false or true whether or not the timeout value was reached or not, respectively. No modern universal programming language supports such a feature as timed-function execution as described here. The following section will describe how to create this functionality and gives a design pattern that arose while implementing the solution. For details on how this was implemented in the worker node and server, refer to Appendix A.2 and A.3, respectively.

4.3.3.1 Creating the Timed Function Execution Pattern

There are really two possible solutions to the problem of timed-function executions, and which one is used is dependent on the nature of the function that requires time-limited execution. The first option is the simpler of the two, both conceptually and to implement. However, it is also more limited in that it only works for a small set of functions, ones that rely heavily on a looped body. Consider a `contains()` method on some generic data structure with the following signature:

```
public boolean contains(Object template, int timeout)
```

This function will return `true` if the object is found in the data structure in the allotted time, and `false`, otherwise. One way to implement this function would be to loop through each item (assuming it is unsorted; otherwise, a binary search would be used) looking for a match. After each comparison the clock will have to be checked to see if there is time remaining to continue searching. Assuming an array data structure, an example implementation is:

```
long start = System.curentTimeMillis();
long current;
for(int i = 0; i < array.length; i++)
{
    current = System.currentTimeMillis();
    if(current - start >= timeout)
        return false;
    if(template.equals(array[i]))
        return true;
}
return false;
```

While this serves the purpose of only continuing to search if time remains, it has two distinct disadvantages:

1. It only works for functions where the main body is inside a loop, or is recursive.

2. A large amount of processor time is wasted by continually checking the clock to see if the function can continue.

The second option that can be used to create a timed function is *similar* to turning it into an asynchronous function call. Two paths of execution are required for this to work (either threads or processes) and they must have a way to communicate data between them. For the purpose of this discussion, we will assume threads are being used. The primary thread is the one that calls the timed function; however, this function is merely a stub used to create the second (worker) thread. The primary thread then goes into a wait state, where the amount of time to wait is the same as the timeout value for the function. The worker thread first initializes a shared-memory area to contain the return value if the function fails; e.g., `false`, or -1. The worker then proceeds to execute the *actual* timed function. When it is complete, the result is placed into the shared-memory area, and the worker thread then notifies the primary thread to wakeup. The worker thread then enters a sleep state. The primary thread, when awakened, returns the value in the shared-memory space as the results for the stub function. If the primary thread

was awakened *early* by the worker, it means the function completed in time and the real value exists in the shared-memory area. If the primary thread woke up on its own (the wait timed out), it means the shared-memory area will still contain the default/failure value. The worker thread will *always* receive a message from the server. If the message is late arriving the worker thread will still enter its sleep state after receiving the message. **Note:** it is unimportant to the primary thread which method was used to wake it up. This solution to timed-function execution has several advantages over the previous solution:

1. There is no wasted CPU time by continually checking the clock to see if the function can continue.

2. The actual function is cleaner, because it doesn't contain the extra code related to timing the function.

3. It can be applied to a broader range of functions, not just those that have looped or recursive bodies.

This solution does have the following disadvantages when compared to the first solution:

1. It requires some standard thread/process capabilities that might not always be available.

2. There is a small overhead for creating the worker thread (more if processes are used). However, the time required for creation should ideally be less than the time wasted by continually checking the clock as in the first solution. The problem could also be further reduced by using a thread pool (*see* Section 4.1.1).

3. It can be conceptually more difficult to understand than the first solution because parallelism is required.

As mentioned earlier, each solution has its place, and in fact *both* solutions are used in the JOLTS system. The first, non-threaded solution is used in the server, while the second, threaded solution is used in the worker node. For details on how this pattern was implemented in the worker and server, refer to Appendices A.2 and A.3.2, respectively.

4.4 Server Side Modules

The JOLTS server side modules are by far the largest modules in the system. The server consists of four separate implementations of the Reactor pattern, and unlike the worker node implementations, all four are required for proper operation of the server. Because of the large number of different types of requests, each group/type of request is handled by a different `Handler` child, as shown in Figure 4.9. Each of the children are in their own sub-modules with all their relevant support classes. Each of these child handlers will now be discussed in turn.

4.4.1 File Handler

The `FileStreamHandler` class is by far the simplest handler on the server. The default configuration of the JOLTS system doesn't permit user jobs to access local files. Any input files required by the job must be submitted along with the job. A special stream class, `GridInputStream`, can be used to access the input files submitted along with the job. The files are stored on the server, but when a request is made to read data from the file by the job executing on a worker node using a `GridInputStream` object, a request is sent to the `FileStreamHandler` object. This handler determines what file is to be read, how much data to read, and from where in the file the data is to be read. The requested data is sent back to the user's job and the handler returns to listening for more incoming data requests. This process creates the illusion of reading local files to the user's job.

4.4.2 Worker Node Handler

Every worker node that registers with the server as being willing to host one (or more) user jobs has a corresponding record created on the server that stores information about the worker node. These records not only keep track of what job(s) the node is currently executing, but also contain general information about the node. This information is used by the scheduler when selecting worker nodes to host submitted jobs. Figure 4.10 contains the `WorkerNodeRecord` class with all its fields and methods.

The `WorkerNodeHandler` class is used to receive messages from worker nodes. As a proper Reactor, the actual *processing* of the message is done by another thread. The type of incoming message determines which class is responsible for processing the message;

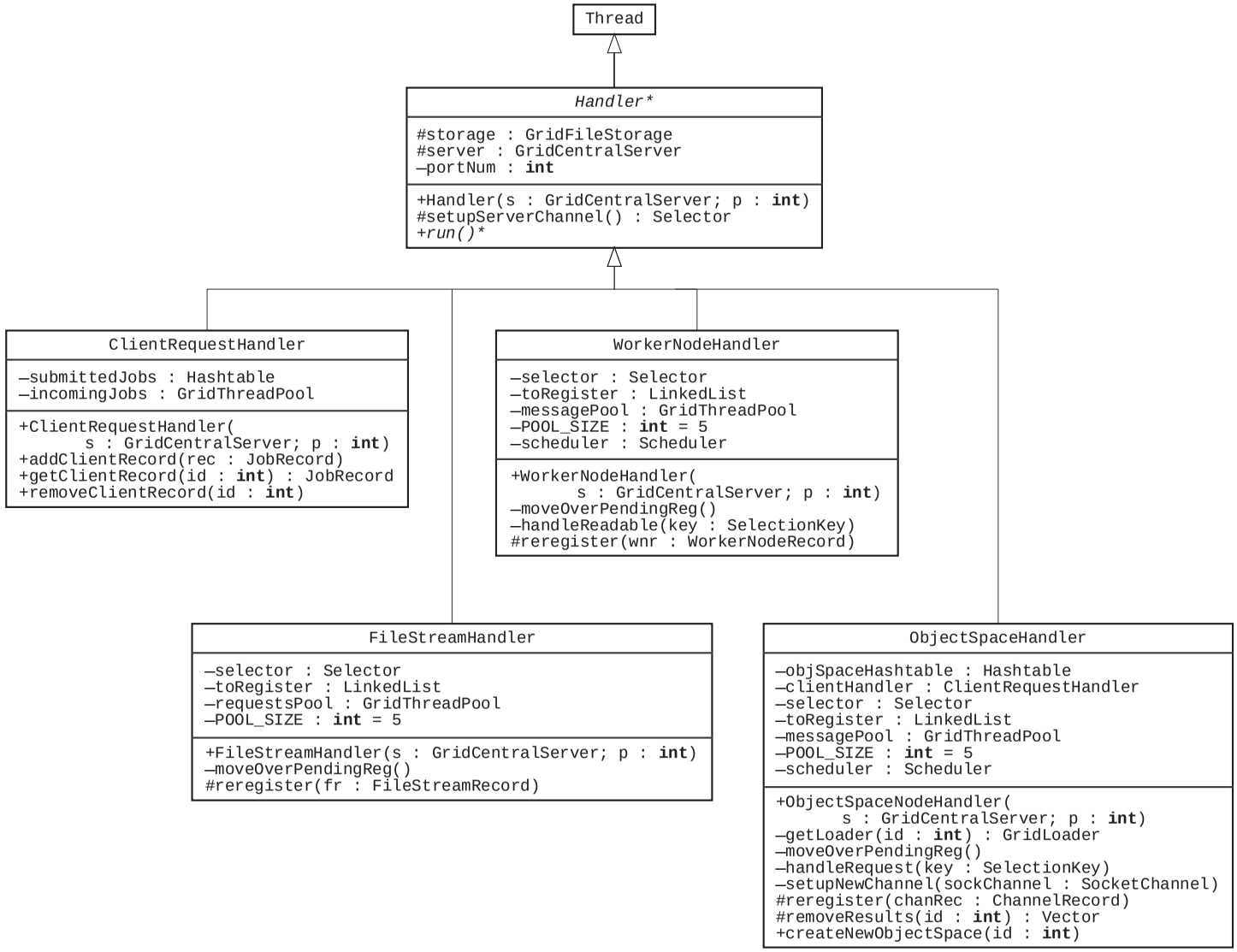

Figure 4.9: The `Handler` hierarchy

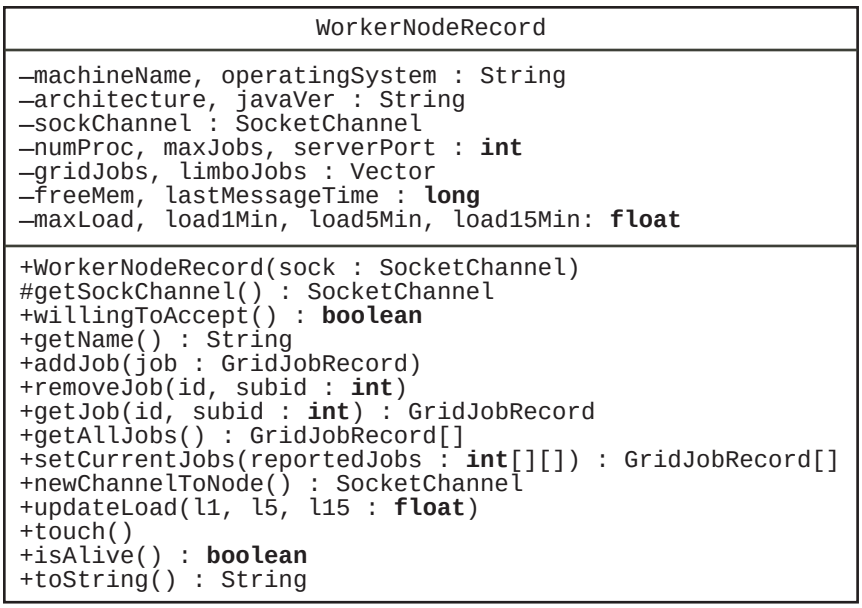

Figure 4.10: The `WorkerNodeRecord` class

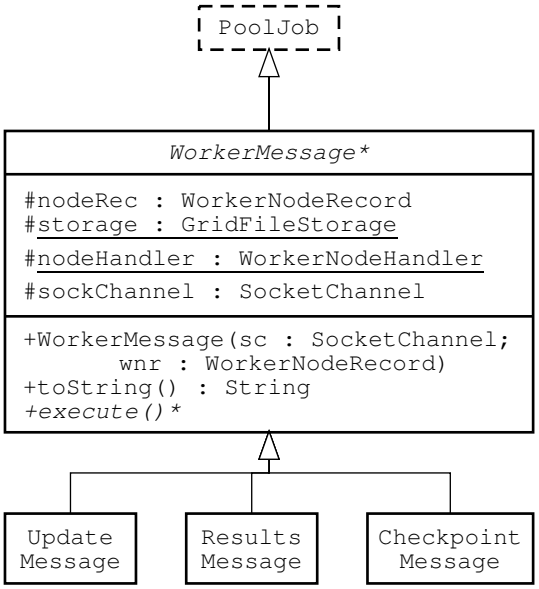

Figure 4.11: The `WorkerMessage` hierarchy

however, it will always be a child of abstract class `WorkerMessage` as given in Figure 4.11. Processing each request in a separate thread, using a thread pool, allows the server to handle incoming data from multiple worker nodes simultaneously.

The `UpdateMessage` class is responsible for updating the statistics of the worker node's corresponding record, so the scheduler can make informed decisions. The incoming message will have originated from a `Heartbeater` object discussed in Section 4.3.1. The

`ResultsMessage` and `CheckpointMessage` instances are used to update the record of a specific job, and to store the incoming data to disk on the server. This incoming message will have originated from the `ReturnResults` class discussed in Section 4.3.2.

4.4.3 Client Handler

The `ClientRequestHandler` class is another implementation of the Reactor pattern designed to handle incoming requests from instances of `GridClient`. Only two types of requests are possible: those submitting a new job to JOLTS, and those wanting to retrieve the results of a previously submitted job, handled by instances of `IncomingCodeJob` and `RetrieveResultsJob`, respectively. The `RetrieveResults` class is much simpler, so it will be described first.

When the results for a job are requested there are only three possible options:

1. There is no job matching the ID supplied by the user. In this case an error message is returned stating the problem.

2. The job hasn't completed yet. In this case a message is returned informing the user to try again later for the results.

3. The job is complete and the results are returned to the client. All traces of the job; e.g., records and temp files, are deleted from the server.

Options 1 and 2 are straightforward, but the last option requires more elaboration.

Most grid systems that support job submissions return some type of output file (usually plain text) as the results for a job. This was deemed infeasible for JOLTS because many of the job types supported consist of many smaller sub-jobs (MISD, SIMD, and Object Space). How to unify the results of the various sub-jobs into one file is not clear, and may not be the best option for the user receiving the results. This problem is even more difficult for object space jobs, since each object space might have a clearly-defined ordering. User programs could potentially create the output files themselves; however, writing files is not supported in JOLTS (by default) because it can potentially cause damage to the worker nodes. As a result, the JOLTS system returns objects as program results. The type of object(s) returned is dependent on the programmer who wrote the job.

For simple/sequential jobs, the completed job is sent back as is. For MISD and SIMD jobs, the results are the job descriptor defined by the programmer as being a descendant of `MISD` or `SIMD`, respectively (*see* Figure 4.6). This descriptor will contain the results of each sub-job in its proper location according to the sub-job ID number. Because each sub-job is in fact a simple/sequential job, this is an efficient way to return multiple results. For object-space jobs, the contents of the object space is returned in a multiset, *not* the active objects that used the object space. (If the active objects are needed, they can just be placed in the object space before they finish executing.) Thus, any important results need to be stored in the object space so they can be retrieved with the results. While this may seem complicated, in reality processing a `RetrieveResultsJob` request is very simple: a serialized object is read from disk and written to a socket channel to return the results. Submitting a new job for execution, on the other hand, is a very complicated process.

New incoming jobs are processed by an instance of `IncomingCodeJob`. The very first thing done is to receive the job's binary description (either a class or jar file) and run it through the JOLTS security manager. This security manager is designed to protect the JOLTS system from malicious code and enforce restrictions that are not definable directly in the Java language. How the security manager works is discussed in Appendix A.1. For the discussion here, it is important only to know that if a job doesn't pass through the security manager, a message is sent to the client explaining why the job was rejected. If the code passes the security manager, it is deemed to be "safe" and the server can begin deciding how to process the job further. The type of job submitted to the JOLTS system is determined by the security manager and information supplied by the user. Depending on the type of the job, the appropriate child of class `NodeCommunicationJob`, shown in Figure 4.12, is created to farm out the job out to the worker nodes. As can be seen in the figure, farming out an entire SIMD job is done using a second class that is used for individual sub-jobs (`FarmOutSIMDJob`). The same holds true for MISD jobs, except that the class used to farm out the individual sub-jobs is the same as the one used for simple/sequential jobs, `FarmOutGridJob`. One important design decision when farming out jobs is how to deal with the multiple parts of MISD/SIMD jobs. Ideally there would be enough available worker nodes to handle all the sub-jobs at once, but this is unlikely to happen when the job has a large number of sub-jobs. This means any remaining sub-jobs will have to wait for a free node, which can potentially block all further incoming

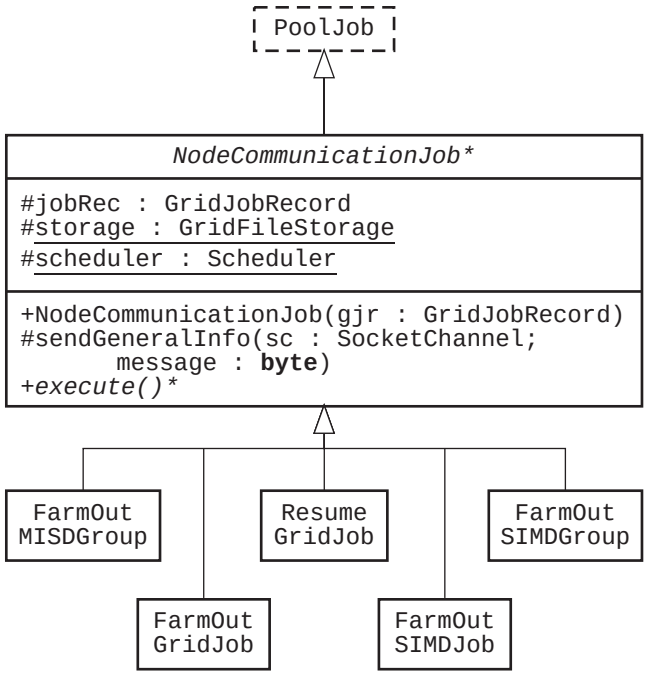

Figure 4.12: The `NodeCommunicationJob` hierarchy

jobs from being executed until all the sub-jobs in a large MISD/SIMD job are completed. This is deemed undesirable, so instead of all the sub-jobs being farmed out to separate threads inside a thread pool, they are all farmed out from the thread responsible for the entire group. Thus, any new job that is submitted while a multi-part job is currently being farmed out has an equal chance of being assigned to the next available worker node. Consider the following example:

> The JOLTS system is running with five worker nodes, each capable of only hosting one job. A SIMD job is submitted that consists of ten sub-jobs. The first five sub-jobs are started, one on each node, while sub-job 6 waits for the next free worker node (jobs 7 through 10 must wait for 6 to be serviced before they can be serviced). At this time a simple/sequential job is submitted to JOLTS for execution. This simple job waits for a free worker node as well, behind sub-job 6, *not* behind sub-job 10. Thus, when the first free node comes up, sub-job 6 is assigned to it, while the second free node will be assigned the simple job, *not* sub-job 7. In this manner smaller sub-jobs are able to squeeze into the waiting queue in the middle of large multipart job.

Along with each job submitted for execution, the server creates a record to store in-

formation about the job, and each of its sub-jobs. This record hierarchy can be seen in Figure 4.13. The only odd thing to note about this hierarchy is the aggregation between the `Collector` and `MultipartGridJobRecord` classes. This is in fact a bidirectional relationship, as can be seen by the presence of the `collector` field in the `MultipartGridJobRecord` class. The result is that each item knows about its enclosing container. While this does go against standard software engineering practices, it is done for several reasons:

1. The collector is responsible for knowing about all its sub-jobs, and properties of the job as a whole.

2. The sub-jobs keep track of their own data (e.g., sub-job ID) while other data, such as the worker nodes that have host any of the sub-jobs, can only be determined by the collector.

All of this record keeping is entirely hidden from the user. However, user-supplied classes are incorporated into this hierarchy in the `descriptor` field of the `Collector` class and the `params` field in the `SIMDGridJobRecord` class.

4.4.4 Server Object Space

The final module in the server to be discussed is the object space sub-module. It was initially suggested that the object spaces be hosted on another machine than the primary central server, or at least in another process on the central server. This would be done for both stability/redundancy reasons and to allow for more resources to be made available for the object spaces. The current version of JOLTS is *not* implemented this way because of the large interprocess communication required between a central server and a separate object space server; however, hooks are present to make adding this functionality in the future easier if it becomes a requirement.

Just like the client and worker modules on the server, this module also contains an implementation of the Reactor pattern in the form of class `ObjectSpaceHandler` (*see* Figure 4.9) and the `ObjectSpaceRequest` hierarchy, shown in Figure 4.14. When a request arrives that wishes to perform an operation on an object space, the appropriate child instance of class `ObjectSpaceRequest` is created to process the request, and is executed

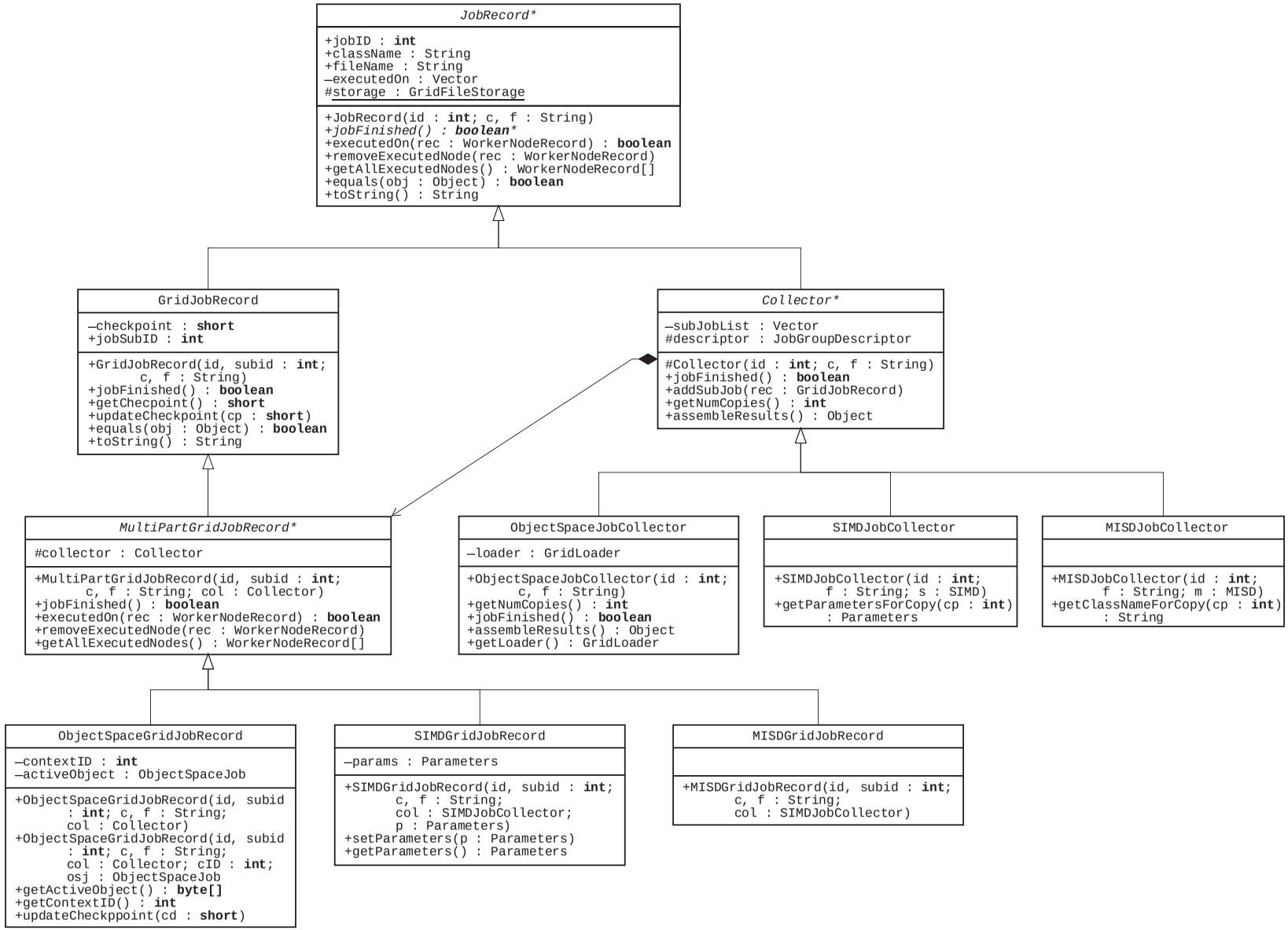

Figure 4.13: The `JobRecord` hierarchy

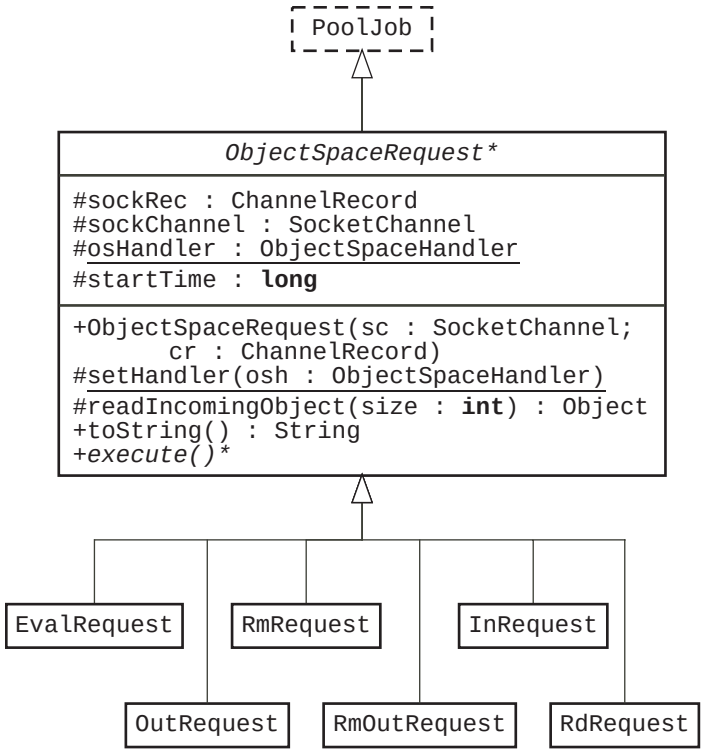

Figure 4.14: The `ObjectSpaceRequest` hierarchy

from inside a thread pool. All of the children, with the exception of `EvalRequest` instances, operate on an instance of `ConcreteObjectSpace`, whose implementation is discussed in Appendix A.3.2.

The reason that an `EvalRequest` instance doesn't operate directly on an object space is that its responsibility of creating active objects doesn't require any manipulation of the actual object space. When an `eval()` request arrives, each object in the multiset is turned into an active object and its "self" object space is created along with a corresponding record (class `ObjectSpaceGridJobRecord` in Figure 4.13). If all the objects were created in the allotted time, the active objects are farmed out to worker nodes for processing.

Note: It is important to realize that while these active objects are being farmed out, there is no guarantee they will begin executing right away, since it is possible that there are no free worker nodes currently available. The Boolean result of the `eval()` method only indicates that the active objects where *created* successfully, *not* that they have all started executing. If the programmer wishes to know when the active objects are actually executing they can create their own message objects to be used

in the object space for confirmation.

The next chapter will give several example programs for the JOLTS system. Included with these example programs will be a discussion of how the system was tested, as well as experimental performance results.

Chapter 5

Empirical Evaluation of the JOLTS System

The JOLTS system was designed according to the goals listed in Section 3.2, and reproduced here for convenience.

1. The system should keep overhead to a minimum.
2. The system should make efficient use of the available resources.
3. The system should be scalable.
4. The system should support several different parallel/concurrent programming models.

This chapter examines if these goals have been met using the Goal Question Metric (GQM) methodology from [2]. The first step in GQM is to convert the goals into questions that can be easily measured. Experiments can then be created to answer these questions. Each section in this chapter focuses on determining if a specific goal has been met.

5.1 Resource Usage

While JOLTS is designed to use the collective resources of many computers to solve any submitted job, it is also important to not needlessly waste these resources. Chapter 3 described the design decisions made to help keep resource usage to a minimum. Chapter 4 described how the high-level design was converted to a detailed design. The three main resources that were carefully considered when designing JOLTS are:

1. CPU — any periodic waiting required is done using passive waits, as opposed to active waits. This mean more CPU cycles are available for the jobs executing on the worker nodes, and searching through the object space on the server.

Table 5.1: Computer Specifications Used for Initial Experiments

Properties	**Worker Nodes**	**Server**
Model	Dell - n Series	Sun Fire 3800
OS Version	Mandrake 9.2	Solaris 8
Processor	Pentium 4 (HT)	Ultra Sparc III (x4)
Processor Speed	2.4GHz	750MHz
RAM	512MB	8 GB
Disk Type	ATA 100	SCSI Ultra2 Wide LVD (RAID 1)
Network	100Mbs	100Mbs
Java Version	1.4.2_02-b0	1.4.2_04-b0

2. Network — all the protocols are byte-based because they require less bandwidth than text-based protocols. Where appropriate, caching was used to prevent possible transmission of redundant data. Also tied closely to the previously mentioned idea of only using passive waits, most of the JOLTS networking also runs on push technology. This reduces the amount of time that the CPU is waiting for data to arrive over the network, as is the case when using pull technology.

3. Disk — although caching is used to keep network usage to a minimum, only the most recent data is cached. For example, if a job is generating checkpoints faster than they can be transmitted (only occurs in non-blocking mode), any checkpoints waiting for transmission are deleted if a newer checkpoint is created before the older checkpoint begins transmission to the server. This results in only the most recent checkpoint being cached to disk.

It would be difficult, if not impossible, to find the *optimal* balance between using these resources. Additionally, the optimal balance would be highly dependent on the nature of each job submitted to run on JOLTS.

5.2 Setup

The various experiments were run at the University of Saskatchewan using the computer science undergraduate computer labs. Worker nodes consisted of standard desktop machines, available for undergraduate use, while the server machine was a dedicated server with restricted access. The specification of the worker nodes and server are given in Table 5.1. While only Linux machines were used as worker nodes during the initial testing, any of the JOLTS components can run anywhere Java 1.4 is available.

Standard output for the server was redirected to `/dev/null`, while standard error was written to the terminal. Output from each worker node, both standard out and standard error, was redirected to a file on a network file system. This was done so that progress could be checked during the experiments. Both the server and worker nodes were started remotely over SSH using scripts. Nodes were added to the grid based on the number required for the current experiment; i.e., all the nodes were *not* started at the beginning of the experiment session. Nodes were set to execute a single job at a time.

The experiments were run late at night (spread over several weeks) to help guarantee network traffic was low, and also to help ensure the worker nodes would not be currently in use by any students. If a worker node was currently in use, it was not used; however, this was impossible to guarantee *completely*, so small fluctuations in results can be expected. If large deviations from the expected results were obtained, the problem node(s) were eliminated from the grid and the corresponding experiment was rerun. When results were requested from the system, a single request was made, instead of repeatedly querying the server. This helps to reduce the server load. Only one job was submitted to the server at a time.

5.3 Overhead

Overhead is inevitable in a system such as JOLTS. The goal of keeping the overhead to a minimum is subjective, because it is hard to define what the *minimum* would be. Overhead can also be dependent on the program that is being measured. Creating a question about the overhead for JOLTS is thus based on the notion of keeping the overhead below some acceptable margin. Without a clear definition of what is deemed "acceptable" the question that needs to be asked is:

> What is the overhead penalty for using the JOLTS system?

To answer this question two programs were designed and run on the JOLTS system with various configurations. The programs run on JOLTS are different approaches that can be used to determine RSA private keys from known public keys. These approaches were chosen because they are simple to parallelize, and they match the programming models offered by JOLTS.

5.3.1 RSA

RSA encryption is based on the notion that it is computationally expensive to factor large numbers [1]. Without going into depth on how RSA works, the process of breaking RSA encryption is to take a *large* value n (public) and determine what its factors, p and q (both private), where both p and q are prime numbers. Using a second public value k, it is then possible to determine all the values required to do RSA encryption and decryption.

At its heart, this is an $O(2^{n/2})$ problem — there is no known polynomial solution (yet). There are many different algorithms, and various optimizations, that can be done to try and factor large prime numbers. Certain algorithms only work once the key sizes pass a certain number of bits, or they rely on special hardware instructions. Because this thesis is not about encryption, the approach taken here is the simplest approach.

1. Find the square root of n, and take the floor of the result.
2. Break the numbers from 1 to the square root into small pieces.
3. The prime numbers in each piece must then be checked to see if they are a factor of n.

In normal RSA use, the size n is usually several hundred to several thousand bits in length. In the example programs used here, the size of n is only 56-bits. The following two subsections describe test programs run on JOLTS that attempt to find p and q using only n.

5.3.2 SIMD Example

This subsection describes a SIMD approach that can be used to break RSA keys. Because this example is designed to be an SIMD program, three main components are needed, the collector, worker, and the parameters. The collector is responsible for creating all the RSA key values, $sqrt(n)$, and breaking the key space into the appropriate number of pieces for the experiment. The worker is assigned a piece of the key space and checks the prime numbers in that key space against n. The parameters are the data used to create the worker. The source code for the collector and worker are in Figures 5.1 and 5.2, respectively. The parameters aren't shown because it contains no interesting methods.

```
import clientside.*;

import java.math.*;
import java.util.Random;

/** This class is responsible for assembling all the sub-jobs as they are completed by the
    worker nodes. */
public class RSACollector extends JobGroupDescriptorTemplate implements SIMD
{
    /** The various values used in RSA encryption. */
    private BigInteger p, q, n, k, phi, d, sqrt;

    /** Variables for recording when the job starts and finishes. */
    private long startTime, endTime;

    /** The constructor used to initialize all the RSA values. */
    public RSACollector()
    {
        super(400); // the number of pieces
        BigInteger message, p1, q1, gcd;
        startTime = System.currentTimeMillis();
        Random r = new Random(10);
        message = new BigInteger("20");

        // calculate all the RSA key values
        p = BigInteger.probablePrime(27, r); // 27-bit prime number
        q = BigInteger.probablePrime(28, r); // 28-bit prime number
        n = p.multiply(q); // will be the combined bit lengths of p and q
        sqrt = sqrt(n).toBigInteger();
        p1 = p.subtract(BigInteger.ONE);
        q1 = q.subtract(BigInteger.ONE);
        phi = p1.multiply(q1);
        do
        {
            // k must be relatively prime to phi AND m^k > n
            k = BigInteger.probablePrime(14, r); // 14-bit prime number
            gcd = phi.gcd(k);
        }while(!(gcd.equals(BigInteger.ONE)
                        && message.pow(k.intValue()).compareTo(n) > 0));
        d = k.modInverse(phi);
    }

    /** Simple method to store the results for a specific sub-job.
        @param obj The results for the sub-job.
        @param cp The number for the sub-job. */
    public void storeResults(Object obj, int cp)
    {
        super.storeResults(obj, cp);
        endTime = System.currentTimeMillis();
    }

    /** Simple method to get the name of the class that represents the sub-jobs.
        @return A String containing the name of the sub-job class. */
    public String getStartingClassName()
    {
        return "RSAWorker";
    }
```

Figure 5.1: The SIMD collector for RSA keys (part 1)

```
    /** Simple method to get the parameters for a specific sub-job.
        @param cp The number of the sub-job to get the parameter for.
        @return The parameter for the specified sub-job. */
    public Parameters getParametersForCopy(int cp)
    {
        BigInteger temp = sqrt.divide(BigInteger.valueOf(results.length));
        BigInteger start = temp.multiply(BigInteger.valueOf(cp));
        BigInteger end = start.add(temp).subtract(BigInteger.ONE);
        return new Range(n, start, end);
    }

    /** This method is used to find the square root of the parameter (since one isn't provided
        for the BigInteger class). It only goes to 15 decimal places of accuracy. */
    private BigDecimal sqrt(BigInteger n)
    {
        BigDecimal newN = new BigDecimal(n);
        BigDecimal guess = new BigDecimal("1");
        BigDecimal two = new BigDecimal("2");
        BigDecimal oldGuess = guess;
        while(true) // intentional infinite loop
        {
            BigDecimal f = oldGuess.multiply(oldGuess);
            f = f.subtract(new BigDecimal(n));
            BigDecimal fPrime = oldGuess.multiply(two);
            guess = oldGuess.subtract(f.divide(fPrime, 15, BigDecimal.ROUND_HALF_DOWN));
            BigDecimal test = guess.multiply(guess);
            if(newN.compareTo(test) == 0) // dead match
                return guess;
            if(guess.equals(oldGuess)) // no movement in guess
                return guess;
            oldGuess = guess;
        }
    }

    /** Simple method to display the main key values created, and the discovered factor.
        @return The calculated keys, and the key discovered by the worker nodes. */
    public String toString()
    {
        String temp = "Keys\n----\n";
        temp += "p: " + p "\n";
        temp += "q: " + q "\n";
        temp += "time: " + (endTime - startTime) + "\n";
        for(int i = 0; i < results.length; i++)
        {
            RSAWorker worker = (RSAWorker) results[i];
            if(worker.getAnswer() != null)
                return temp + "calculated factor: " + worker.getAnswer() + "\n";
        }
        return temp;
    }
}
```

Figure 5.1: The SIMD collector for RSA keys (part 2)

```
import clientside.*;
import gridutil.CheckpointMech;

import java.math.*;

/** This class is responsible for looking for a factor for n inside a specific range of values.
    The range to be searched is passed into the constructor. */
public class RSAWorker implements GridJob
{
    /** Instance variables for designating the range to be searched, n, and the found factor. */
    private BigInteger start, end, n, answer;

    /** Simple constructor used to set the instance variables, taken from the paramter to the
        constructor. */
    public RSAWorker(Parameters p)
    {
        Range r = (Range) p;
        n = r.n;
        start = r.start;
        end = r.end;
    }

    /** Empty method, checkpoint mechanism isn't being used. */
    public void setCheckpointMech(CheckpointMech cpm) {}

    /** Simple method to get the answer found by this object. */
    public BigInteger getAnswer()
    {
        return answer;
    }

    /** The heart of the worker. It tries to find a factor of n in the range between start and
        end. If it found a factor, it stores it in an instance variable. */
    public void execute()
    {
        BigInteger p = factor(n, start, end);
        if(!p.equals(BigInteger.ZERO)) // found factor
            answer = p;
    }

    /** This method tries to find the factors for a given number. It only checks for a factor
        inside a given range. If it isn't found, it returns 0. */
    private BigInteger factor(BigInteger n, BigInteger start, BigInteger end)
    {
        BigInteger two = new BigInteger("2");
        BigInteger current = start;
        if(current.remainder(two).equals(BigInteger.ZERO)) // make sure the number is odd
            current = current.add(BigInteger.ONE);
        while(current.compareTo(end) <= 0)
        {
            if(current.isProbablePrime(100)) // 0.9990234375 certainty
            {
                BigInteger rem = n.remainder(current);
                if(rem.equals(BigInteger.ZERO))
                    return current;
            }
            current = current.add(two);
        }
        return BigInteger.ZERO;
    }
}
```

Figure 5.2: The worker for breaking RSA keys

The server was configured to handle 15 parallel worker node requests, and the heart beat length was set at 5 seconds. The Java bytecode for the job was 3.5K in size. The results of this program with increasing number of nodes are given in Figure 5.3. The data used to created the graph can be found in Appendix B. An explanation of each line in the graph is as follows:

1. Standalone — This version was run on a worker node, but *not* as part of the JOLTS system. This was done to determine the amount of time required by a single-threaded version, with no system overhead. This line is horizontal because it is a base line for comparison *outside* the system.

2. Thread Ideal — Because this program can easily be decomposed into parallel pieces, this line represents the ideal case of *no overhead* when running multiple threads. A special run of the program was made to determine the time required to search the *entire* key space. The ideal value for parallelizing the program is to take this time (approximately two hours), and divide it by the number of threads/nodes. As can be clearly seen, there are diminishing returns as more threads are used, even with no overhead.

3. Threaded Local — Again, since this program can easily be decomposed into parallel pieces, it might seem natural to run each piece in its own thread on a single machine. This line represents the program run with multiple threads, all on the same machine. The number of threads used corresponds to the number of worker nodes in the graph.

4. Blocks = Nodes — This version was run on increasing numbers of worker nodes *inside* the JOLTS system. The key space was divided evenly among the worker nodes, with each worker node assigned a single block to process.

5. Blocks = 400 — This version is similar to the preceding one, except the number of pieces the key space was divided into was fixed at 400 pieces. As a node finished its assigned piece, it was assigned another piece to process until all 400 blocks had been processed. This represents the case where the number of parameters experiments to run is much larger than the number of available nodes.

The timing for the experiments run inside the JOLTS system are based on the time between when the job record is created on the server, until the final result is returned by the worker

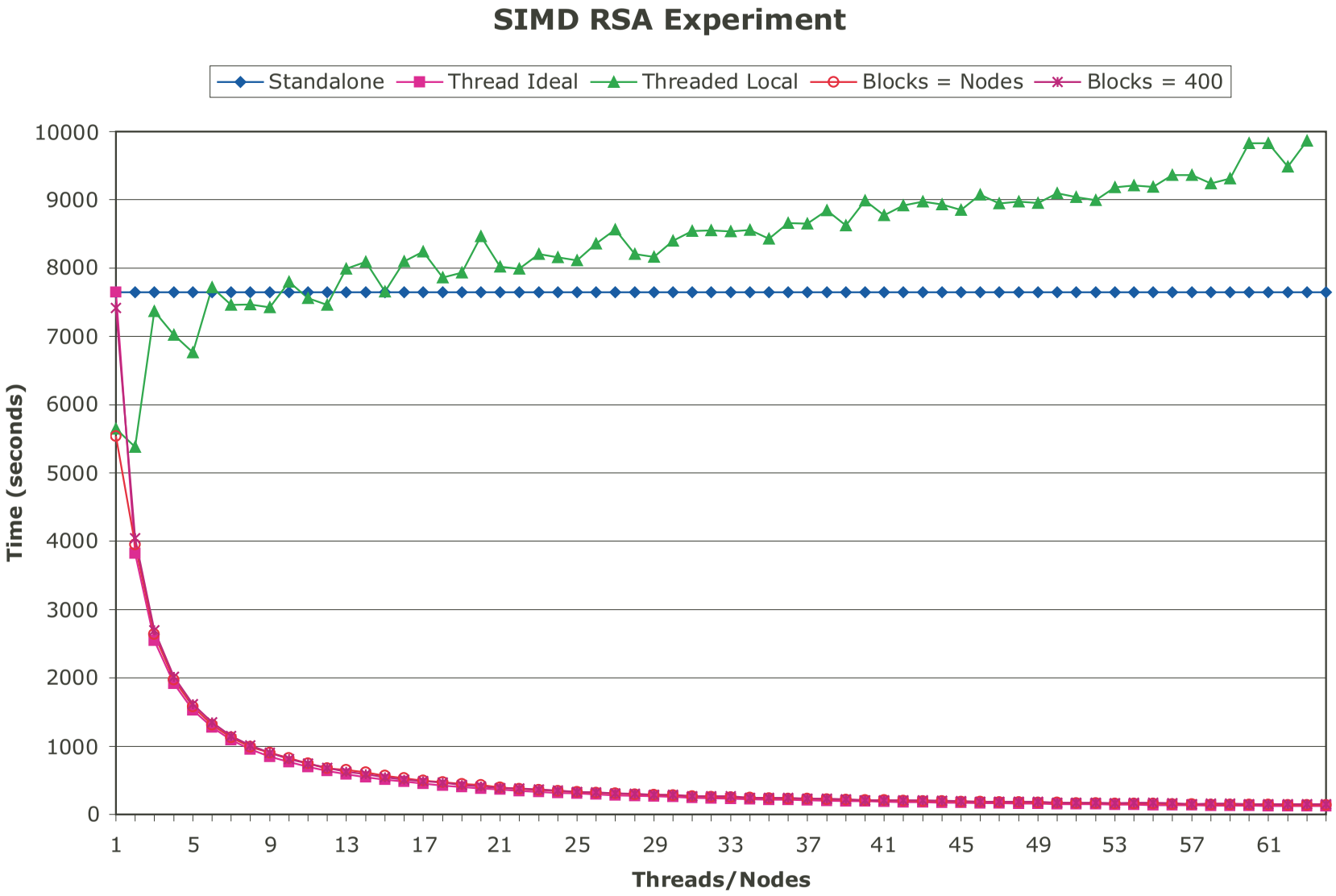

Figure 5.3: The results of the RSA SIMD experiment

node. Normally the results of a completed job are only assembled when the results are requested by the user; however, to facilitate proper timing for these experiments, the results are assembled as a soon as the last sub-job result arrives from a worker node. No checkpoints were created while the program was running.

As can be seen from the graph, the Thread Local line has a general upward trend, when more than two nodes are used. The reason why with only two threads the execution time decreases is that there are still free cycles available when only one thread is used, which this second thread is able to use. With more than two threads present, the processor begins thrashing, decreasing the available time for the threads and increasing the execution time of the program.

The two experiments run using JOLTS, Blocks = Nodes and Blocks = 400, show an obvious trend matching the Thread Ideal line. The gap between the Thread Ideal line and the two Block lines is the overhead incurred when using the JOLTS system. The overhead is very small for this example program as more nodes are added to the grid, indicated by the fact that the lines converge rather quickly. Although it is hard to tell from the graph due to the fact the lines appear to be overlapping, there is still a small measurable overhead. The overhead for this experiment is defined to be

$$overhead\% = \left(1 - \frac{\text{experimental time}}{\text{ideal time}}\right) * 100$$

The minimum overhead reached on the Blocks = Nodes line is 2.98%, obtained with six nodes in the grid. After this point, overhead continues to rise at progressively smaller increments, approaching 12% at the tail end of the graph. While the percentage difference between the ideal theoretical value and the experimental results is growing, the actual value represented by this difference is decreasing throughout the graph.

The minimum overhead reached on the Blocks = 400 line is 5.26%, obtained with four nodes in the grid. After this point, overhead continues to rise, approaching 25% at the tail end of the graph. This line has a larger overhead than the Blocks = Nodes line, which is consistent with the expected results. With more blocks, there is more network traffic as additional blocks are sent to worker nodes that finish their assigned block. While new blocks are in transit to the worker nodes, the worker is sitting idle. This idle time is the cause of the slower program performance. In the Blocks = Nodes line, there is no idle

worker time, because once a worker node finishes its assigned block, there are no more blocks that need to be sent to the worker node.

There are two possible values that can directly affect the execution time on a worker node: the heart-beat length, and whether or not the sub-jobs are sent to the worker nodes in sequence, or in parallel. As mentioned in Section 4.3.1, the heart-beat length is the amount of time between the periodic updates from the worker node to the server. When this heart beater is executing, it consumes CPU cycles that could otherwise be used by jobs running on the worker node. With a smaller value, more CPU cycles are available for running jobs; but conversely, the worker node is then less responsive to changes outside the JOLTS system. The heart-beat length was kept constant at five seconds throughout the experiments.

As mentioned earlier in the example on page 85, it was decided that the sub-jobs that create SIMD/MISD jobs are sent to worker nodes in a sequential manner. This allows smaller jobs equal opportunity to begin execution instead of having to wait for a large multipart job to finish executing first. Thus, rerunning the experiments sending out all the sub-jobs in parallel was not done.

5.3.3 Object Space Example

This subsection describes an object space approach that can be used to break RSA keys. Unlike the SIMD approach discussed in the previous section, communication is possible between sub-jobs using the object space. Thus, once either p or q has been found, no more subsequent blocks from the key space are searched. Blocks that are *currently* being processed are allowed to finish.

One active object places the ranges to be processed in the object space. Workers are responsible for retrieving a range from the object space and checking if p or q is in the given range. If it is not found, another range is retrieved and the process repeats. If the factor *is* found in the range, that active object removes all the other ranges from the object space, and then inserts the results into the object space. If there are no ranges left in the object space when an active object attempts to retrieve a range, the active object just exits. The source code for the starting active object and the workers are given in Figure 5.4 and 5.5, respectively.

The worker nodes and server are the same machines given in Table 5.1. The object

```
import clientside.*;
import gridutil.objectspace.*;

import java.math.*;
import java.util.Random;

/** This class generates all the RSA key values. Some of them are random (e.g., p and q), while others
    are the result of calculations (e.g., n and phi). The keys are placed in the object space. Then the key
    space is broken down into segments, and the ranges for these segments are placed into the object
    space. Finally, several workers are created to process the ranges, looking for the factors of n. */
public class Starter extends ObjectSpaceJobTemplate
{
    /** The number of blocks to decompose the key space into. */
    private final int NUM_BLOCKS = 400;

    /** The number of active objects to process the key space. */
    private final int NUM_WORKERS = 64;

    /** This method first generates the random RSA keys, which are then stored in the object
        space in a 'Key' object. Then the range values for the primes p & q are broken into
        100 blocks, whose ranges are placed in the object space. Finally, some workers are
        created to process the blocks looking for a valid factor for n. */
    public void execute()
    {
        Random r = new Random(10);
        BigInteger message = new BigInteger("20");

        // calculate all the RSA key values
        BigInteger p = BigInteger.probablePrime(27, r); // 27-bit prime number
        BigInteger q = BigInteger.probablePrime(28, r); // 28-bit prime number
        BigInteger n = p.multiply(q); // will be the combined bit lengths of p and q
        BigInteger p1 = p.subtract(BigInteger.ONE);
        BigInteger q1 = q.subtract(BigInteger.ONE);
        BigInteger phi = p1.multiply(q1);
        BigInteger k, gcd, d;
        do
        {
            // k must be relatively prime to phi AND m^k > n
            k = BigInteger.probablePrime(14, r); // 14-bit prime number
            gcd = phi.gcd(k);
        }while(!(gcd.equals(BigInteger.ONE)
                && message.pow(k.intValue()).compareTo(n) > 0));
        d = k.modInverse(phi);

        // place data in object space for processing
        MultiSet m = new MultiSet();
        m.put(new Keys(p, q, k));

        // break range of factors into blocks and insert into object space
        BigInteger sqrt = sqrt(n).toBigInteger();
        BigInteger blockSize = sqrt.divide(new BigInteger(NUM_BLOCKS + ""));
        BigInteger start, end = BigInteger.ZERO;
        for(int i = 0; i < NUM_BLOCKS; i++)
        {
            start = end.add(BigInteger.ONE);
            end = start.add(blockSize);
            m.put(new Range(i, start, end));
        }
        self.out(m, ObjectSpace.INFINITE_TIMEOUT);

        // create workers to process range blocks
        m = new MultiSet();
        for(int i = 0; i < NUM_WORKERS; i++)
            m.put(new Cracker());
        self.eval(m, ObjectSpace.INFINITE_TIMEOUT);
    }
```

Figure 5.4: The class for the starting active object (part 1)

```
    /** This method is used to find the square root of the parameter (since one isn't provided
        for the BigInteger class). It only goes to 15 decimal places of accuracy. */
    private static BigDecimal sqrt(BigInteger n)
    {
        BigDecimal newN = new BigDecimal(n);
        BigDecimal guess = new BigDecimal("1");
        BigDecimal two = new BigDecimal("2");
        BigDecimal oldGuess = guess;
        while(true) // intentional infinite loop
        {
            BigDecimal f = oldGuess.multiply(oldGuess);
            f = f.subtract(new BigDecimal(n));
            BigDecimal fPrime = oldGuess.multiply(two);
            guess = oldGuess.subtract(f.divide(fPrime, 15, BigDecimal.ROUND_HALF_DOWN));
            BigDecimal test = guess.multiply(guess);
            if(newN.compareTo(test) == 0) // dead match
                return guess;
            if(guess.equals(oldGuess)) // no movement in guess
                return guess;
            oldGuess = guess;
        }
    }
}
```

Figure 5.4: The class for the starting active object (part 2)

space was configured to handle five parallel object space requests at a time. The rest of the server values are the same as those given in Section 5.3.2. The Java bytecode for the job was 5.2K in size. The results of this experiment with increasing number of nodes are given in Figure 5.6. The data used to created the graph can be found in Appendix B. An explanation of each line in the graph is as follows:

1. Ideal — This version was calculated based on the time required for a single worker running *outside* of JOLTS to find a factor. This number is then divided by the number of threads/nodes to create the ideal time required if the program was run in parallel, with *no* overhead.

2. Threaded (Blocks = Nodes) — For this version a special object space was created that ran outside of the JOLTS system. It was run on a single machine, while the number of threads (on the same machine) was increased. Each active object processed a single range before finishing.

3. Threaded (Blocks = 400) — This version is identical to the previous one except that the number of blocks to be processed was held constant at 400.

4. JOLTS (Blocks = Nodes) — This version was run on JOLTS, with each worker being

```
import clientside.*;
import gridutil.objectspace.*;

import java.math.*;

/** This class is responsible for trying to factor a large number, finding its two factors (which are
    both prime numbers). This is useful in breaking RSA keys. */
public class Cracker extends ObjectSpaceJobTemplate
{
    /** The keys being processed. Only the public ones are accessed. */
    private Keys keys;

    /** The current range being processed. It is stored here so that it can be checkpointed. */
    private Range range;

    /** The current checkpoint number. */
    private short cpNum;

    /** The heart of the class. It first gets the public keys (n and k). It then obtains a block
        range for processing, looking for a factor of n in the range. This repeats until either there
        are no blocks left, or a factor is found. If a factor is found, all the result is placed in
        the object space, and all the remaining blocks are deleted. */
    public void execute()
    {
        BigInteger n, k;
        MultiSet m = null;
        if(range == null)
        {
            keys = new Keys();
            m = context.rd(keys, 1, 1, ObjectSpace.INFINITE_TIMEOUT);
            keys = (Keys) m.get(0);
            range = new Range();
            m = context.in(range, 1, 1, 5000); // 5 seconds
            range = (Range) m.get(0);
            cpMech.checkpoint(++cpNum);
        }

        n = keys.n;
        k = keys.k;

        while(range != null)
        {
            BigInteger p = factor(n, range.start, range.end);
            if(!p.equals(BigInteger.ZERO)) // found factor
            {
                BigInteger q = n.divide(p);
                Results results = new Results(p, q, k);
                m = new MultiSet();
                m.put(results);
                context.out(m, ObjectSpace.INFINITE_TIMEOUT);
                while(context.rm(range, 1, 5, 3000) == 5)
                    ; // just keep deleting
                break;
            }
            m = context.in(range, 1, 1, 5000); // 5 seconds
            range = (Range) m.get(0);
            cpMech.checkpoint(++cpNum);
        }
    }
```

Figure 5.5: The class for the worker active object (part 1)

```
    /** This method tries to find the factors for a given number. It only checks for a factor
        inside a given range. If it isn't found, it returns 0. */
    private BigInteger factor(BigInteger n, BigInteger start, BigInteger end)
    {
        BigInteger two = new BigInteger("2");
        BigInteger current = start;
        if(current.remainder(two).equals(BigInteger.ZERO)) // make sure the number is odd
            current = current.add(BigInteger.ONE);
        while(current.compareTo(end) <= 0)
        {
            if(current.isProbablePrime(100)) // 0.9990234375 certainty
            {
                BigInteger rem = n.remainder(current);
                if(rem.equals(BigInteger.ZERO))
                    return current;
            }
            current = current.add(two);
        }
        return BigInteger.ZERO;
    }
}
```

Figure 5.5: The class for the worker active object (part 2)

assigned to its own worker node. Each active object processed a single range before finishing. No checkpoints were used.

5. JOLTS (Blocks = 400) — This version was run on JOLTS, with each worker being assigned to its own worker node. The number of blocks was held constant at 400. No checkpoints were used.

6. JOLTS (Blocks = 400 with Checkpoints) — This version was run on JOLTS, with each worker being assigned to its own worker node. The number of blocks was held constant at 400. Checkpoints were created each time an active object retrieved a block for processing from the object space. The size of the resulting checkpoint was 1147 bytes.

The timing for the experiments run inside the JOLTS system are based on the time between when the job record is created on the server, until the final active object exits and the object space is serialized to disk. This resulting value can only be obtained by looking at the server logs — there is no way for the user to accurately determine this value. The code to perform this timing was added only for these experiments, and is not normally present.

As can be seen in the graph, the two "Threaded" lines have a general upward trend. The line where the number of blocks matches the number of nodes rises more quickly because a large amount of processor time is wasted searching unnecessary blocks. Basically,

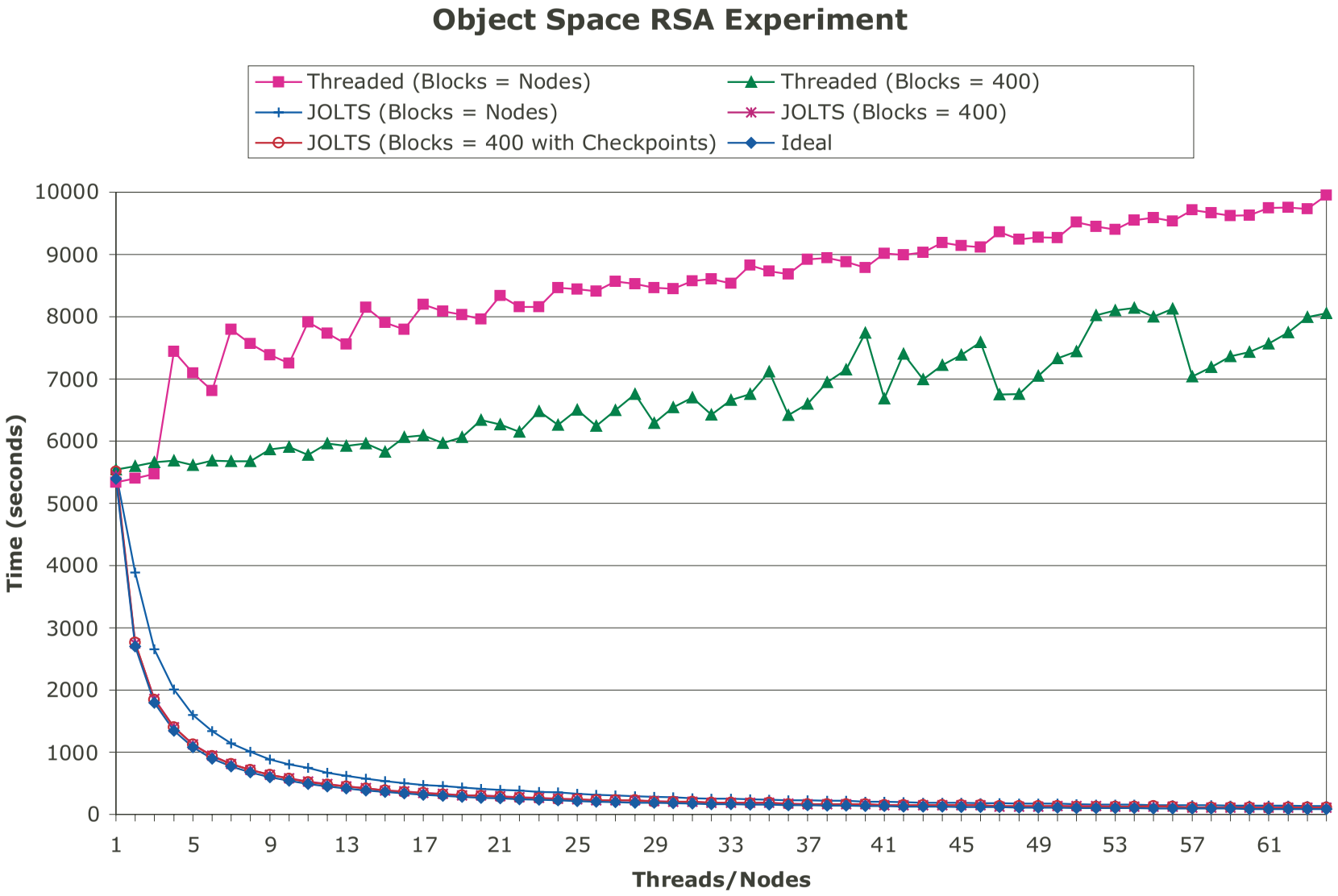

Figure 5.6: The results of the RSA object space experiment

each block will be *fully* searched, except for the single block where a matching factor is found. The second "Threaded" line rises slower because as soon as the factor is found, no other blocks are processed. This saves processor time, resulting in a lower slope to the line.

The three experiments run using JOLTS show an obvious trend matching the "Ideal" line. The experiment where the number of blocks equals the number of nodes is noticeably above the ideal line for the same reason just mentioned in the previous paragraph. The entire key space is being searched, instead of just until a factor is found. This results in an average percentage overhead of 53% over the length of the graph, with only 43% with only two nodes, up to 59% with 64 nodes.

The two experiments where the number of blocks is held constant at 400 virtually overlap the Ideal line. Also, the percentage overhead between these two lines shows little difference between whether or not checkpoints are used. In some experiments the line with no checkpoints is faster, and it others the line *with* checkpoints is faster. The time difference between these two is always within three or four seconds, well within the general time fluctuations between runs. The important thing to realize is that there is basically no performance penalty present in using checkpoints versus not using checkpoints. The percentage overhead for both lines range from 2% overhead to 28% overhead. Again, although the overhead percentage continues to rise, the actual time difference between the experimental runs and the Ideal line continues to get smaller.

While these overhead values may seem high, when using the object space the choice of computer used as the server can have a significant impact on the performance on object space jobs, shown in the following section. As it turns out the server used for the object space experiments in this section was not the best choice. Using a different server can result in faster execution, which will in turn lower the percentage overhead.

5.4 Scalability

Scalability is an important attribute in distributed systems. Every system, no matter what it is, has a bottleneck. Bottlenecks can't be eliminated, they can only be shifted between areas; e.g., from the CPU to the network connection. A grid environment such as JOLTS is scalable if it can handle a large number of nodes. Because all the nodes in

the system need to communicate with the server, the server is a potential bottleneck. In theory, if the number of nodes in the system is large enough, it should have a noticeable impact on performance.

For most job types, the server performs very few calculations. However, for object space jobs, the server can potentially perform a sizeable amount of work while performing various requested operations on object spaces. Thus, to stress test the server, it is ideal to use object space jobs that make frequent calls to the object space. The system can be run using different types of servers, to help determine the type of server required for acceptable performance. For example, is a standard PC just as effective as the JOLTS server compared to a $100,000 mainframe?

The example program used to test the scalability of the system is a slight modification of the RSA program discussed in Section 5.3.3. The modifications made to the program to stress test the server are:

- The checkpoints performed by the active objects are done in blocking-mode. Thus, *each* checkpoint is sent to the server, not just the most recent. This eliminates the possibility of potentially loosing the factor of n to a failed worker node, and also has the effect of keeping the network busy all the time.

- The number of blocks in the key space was increased from 400 to 2000. This reduces the size of each block, and thus the time required to process the block. The worker nodes can process a block in three to five seconds, depending on the type of worker node.

- The checkpoint size was artificially increased. The experiment in Section 5.3.3 used checkpoints of 1K. For the experiment here, the checkpoints are increased in size to 101K. This was done by creating a large `int` array in the active object that needs to be included with the checkpoint data. This stresses not only the network, but also the disk on the server for storing all these checkpoints.

- A total of 5000 dummy objects were placed near the front of the object space. This increases the load on the server, as each block request by an active object must first search through these 5000 blocks, before finding a valid block to be processed. When an active object finds the desired factor of n, it not only removes the remaining

Table 5.2: Server Specifications Used for Stress Testing

Server Name	**Tangra**	**Sprite**	**Penguin/Morph**	**Stealth**
Model	Sun Fire 3800	Xserve G5	Dell - n Series	IBM NetVista
Processor Type	Ultra Sparc III	PowerPC 970fx	Pentium 4 (HT)	Pentium 4
Processor Speed	750MHz (x4)	2GHz (x2)	2.4GHz	1.8GHz
OS	Solaris 8	OSX Server 10.3.4	Mandrake 9.2	Mandrake 9.2
Memory	8GB	2GB	512MB	256MB
Hard Drive	SCSI Ultra2 Wide LVD (RAID 1)	SATA	ATA 100	ATA 100
Network	100 Mbs	1 Gbs	100 Mbs	100 Mbs
Java Version	1.4.2_04-b05 (64-bit)	1.4.2_03-117.1	1.4.2_02-b03	1.4.2_02-b03

Server Name	**Skorpio**	**Glycogen**	**Sobek**
Model	Sun E450	Sun Blade 1000	IBM X335
Processor Type	Ultra Sparc II	Ultra Sparc III	Xeon
Processor Speed	296MHz (x4)	600MHz	2.4GHz
OS	Solaris 7	Solaris 8	Mandrake 9.1
Memory	2GB	512MB	512MB
Hard Drive	SCSI2 (RAID 1)	Fibre Channel	Ultra SCSI 320 (RAID 1)
Network	100 Mbs	100 Mbs	100 Mbs
Java Version	1.4.2_04-b05 (64-bit)	1.4.2_04-b05 (64-bit)	1.4.2_03-b02

blocks from the object space, it also removes these dummy objects five at a time with a three second timeout. This creates a flood over 1000 requests to the server in a very short span of time.

- Two thread pools on the server were increased in capacity to handle the large number of nodes. The `WorkerNodeHandler`'s pool size was increased from 15 to 30 threads. The `ObjectSpaceHandler`'s pool size was increased from 5 to 20.

As mentioned earlier, this experiment was designed to stress test the server. To find a "good" server for JOLTS, this test was run on many different servers. The specifications for the servers and worker nodes used are given in Tables 5.2 and 5.3, respectively. The nodes were added to the system, five nodes at a time, starting with the Penguin/Morph machines, followed by the Stealth, and lastly the Spinks machines. Standard out for the server was redirected to `/dev/null`, while standard error was displayed on the terminal. For worker nodes, both standard out and error where redirected to a file (each node had their own file) on a NFS mount point. For the Windows worker nodes, standard out and error was just displayed to the DOS prompt.

Table 5.3: Worker Node Specifications Used for Stress Testing

Node Name	**Penguin/Morph**	**Stealth**	**Spinks**
Model	Dell - n Series	IBM NetVista	Dell - n Series
Processor Type	Pentium 4 (HT)	Pentium 4	Pentium 4 (HT)
Processor Speed	2.4GHz	1.8GHz	2.8 GHz
OS	Mandrake 9.2	Mandrake 9.2	Windows 2000 Pro
Memory	512MB	256MB	512MB
Hard Drive	ATA 100	ATA 100	ATA 133
Network	100 Mbs	100 Mbs	100 Mbs
Java Version	1.4.2_02-b03	1.4.2_02-b03	1.4.2_02-b03
Nodes used	60	45	35

The results of the experiment are shown in Figure 5.7. The values for experiments with 5 and 10 nodes have values of around 18.5 minutes and 9.4 minutes, respectively. These are not shown in the graph because such large values make the differences present in the tail of the graph less noticeable. The general trend of the graph is a downward curve with diminishing returns, much like several of the lines in Figure 5.6.

The first thing to note about this graph is the data using Sprite as the server only goes up to 100 nodes. The *entire* server would crash when higher number of nodes were attempted, and none of the debugging messages displayed. In an attempt to isolate the cause of the instability using Sprite, tests were run on an older Xserve machine (not shown), and it too exhibited the same instability when high numbers of nodes were used. Thus, it appears to a be a problem with the JVM on that operating system. This is *not* a problem with the JOLTS system, because the exact same server bytecode was used on all the servers, and only those running OSX Server exhibited this instability.

The system using the servers Penguin, Stealth, and Sobek appear to handle the increasing number of nodes without problems. Multiple test runs resulted in only small time fluctuations of approximately 1.5 seconds. As expected, the performance gained by adding new nodes is very small near the tail end of the graph, with speeds increasing only 2 to 3 seconds when adding 5 additional nodes. However, this trend cannot continue indefinitely, eventually the network would become saturated.

Assume that a worker node can process a data block in 3.5 seconds, and that the checkpoints returned by the worker nodes are spread out evenly over those 3.5 seconds.

$$\begin{aligned} \text{Network Speed} &= \text{Nodes} * \text{Checkpoint size}/\text{Checkpoint frequency} \\ 100\text{Mb/s} &= x * 101\text{K}/3.5\text{s} \end{aligned}$$

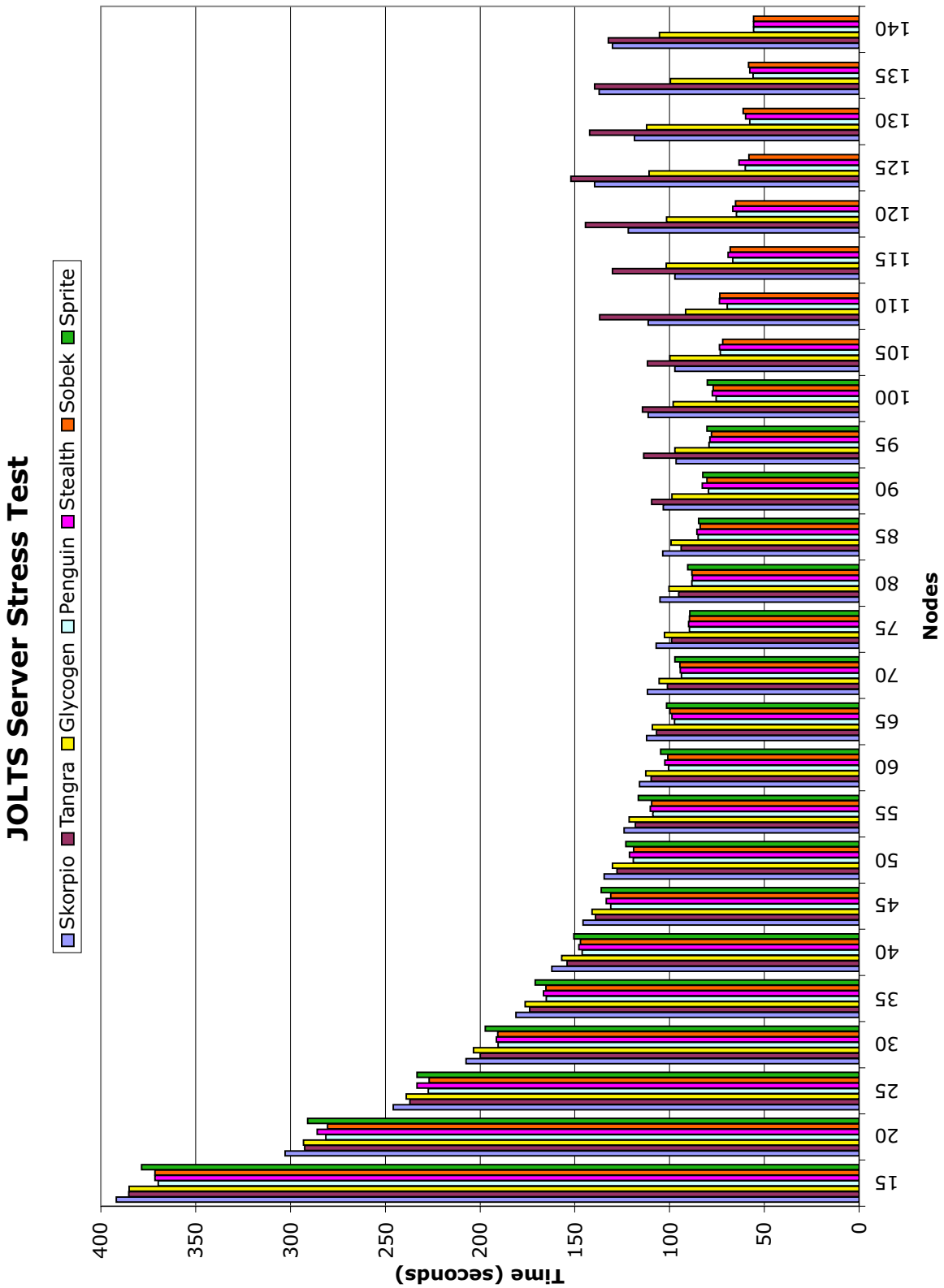

Figure 5.7: The results of the stress test experiment

$$12.5\text{M/s} = x * 0.0289\text{M/s}$$
$$432 \approx x$$

This indicates that in an ideal case, the system would at most be able to handle 432 nodes when using a checkpoint size of 101K before the network became saturated. In reality, it would require fewer nodes because there are other items that lower the available network speed; e.g., TCP retransmissions, heartbeat data, and packet overhead. It would also require a fast server to handle that many concurrent connections.

Looking at the graph, it is apparent that three servers, Skorpio, Tangra, and Glycogen begin *increasing* in time as more nodes are added around 90 nodes. The network cards on these servers appear to be unable to cope with that many concurrent connections, even though the network cards are supposedly rated at 100Mbs. When the network cards become saturated, the server is unable to process worker node requests to the object space before the timeout value has been reached (3 seconds for this example program). When a node times out, it no longer contributes any of its cycles to the job being processed. For example, with 100 nodes in the system, if 30 nodes reach their timeout value, it in essence becomes a 70 node grid. Based on how many nodes timeout, it can drastically affect the results for a test run. Because nodes are bursting data at 101K, as opposed to an even stream of data, this bursting can cause a large number of nodes to timeout at once. This behavior was observed by examining the files in the JOLTS "temp" directory on the server. When an active object finishes, a results file is sent to the server. Normally these results files only appear when a sub-job is finished; but when an active object times out, the results file is sent to the server well before the job finishes, which *did* occur on the servers where the execution time is increasing. The values given in Figure 5.7 are the best/fastest recorded times. For the Skorpio and Tangra servers, with large number of nodes, the fluctuations between test runs easily approach two minutes in variance. Glycogen, the other server that performed poorly when using lots of nodes also had fluctuations between runs, but only around 50 seconds in variance.

This experiment shows that one of the most important factors for the JOLTS server is the network interface. If object space requests can't get to the server fast enough, it doesn't matter how fast the server is, the requests will timeout. Once the request does manage to reach the server, the faster the CPU, the better. The disk speed on the server

appears to have little influence, since even with the 140 nodes all checkpointing at the same time, it is easily handled by the disks in the servers being tested.

5.5 Programming Models

One of the goals of the JOLTS system was to support many different programming models. This goal can easily be converted into a question:

> Does the JOLTS system support multiple programming models?

Answering this question doesn't require any detailed experiments, merely a discussion of the features supported in JOLTS. JOLTS supports several different programming models, some directly and some indirectly. The sequential model is the simplest to support and is done so using the `GridJob` interface. This interface is also used in supporting both MISD and SIMD programming models. Each of the remaining subsections discusses a different programming model supported by JOLTS.

5.5.1 SIMD/MISD

SIMD and MISD models depend strongly on the ability to specify the multiple components that make up a single job. This is supported using the `SIMD` and `MISD` interfaces, respectively. An example of using `SIMD` was given in Section 5.3.1. As mentioned earlier, there are many different algorithms that can be used to do prime number factorization. Each of these algorithms could be created in a separate class and assembled into a MISD job. Thus, the various algorithms (multiple instructions) could be compared to see how long they take to break the same key (same data). Figure 5.8 contains an example class that could be used to describe such a MISD job. Each of the String names stored in the array `names` correspond to a different class (not shown) that implements the corresponding algorithm.

5.5.2 Object Space

The most interesting programming model supported by JOLTS is the object-space model. Using the object space, programs can be designed using either high-level models such as shared memory, or low-level models using semaphores. It is even possible to emulate both

```
import clientside.*;

/** This class is responsible for describing a MISD grid job that consists of several different
    classes, each using a different algorithm for factorization of prime numbers. */
public class FactorizationAlgs extends JobGroupDescriptorTemplate implements MISD
{
    /** An array for storing the names of all the various classes making up this MISD job. */
    private String names[ ];

    /** Simple constructor that stores the name of the various sub-jobs in the 'names' array. */
    public FactorizationAlgs()
    {
        super(4);
        names = new String[results.length];
        names[0] = "BruteForce";
        names[1] = "NumberSieve";
        names[2] = "QuadraticSieve";
        names[3] = "PollardStraassen";
    }

    /** Simple method to get the name of the class for a specific sub-job of this MISD job. */
    public String getClassNameForCopy(int cp)
    {
        return names[cp];
    }
}
```

Figure 5.8: An example class for various prime number factorization sub-jobs

unicast and multicast message-passing programming models using the object space. As soon as multiple entities are able to communicate and coordinate with each other, many different programming models are possible. The most common use for object spaces is the creation of master/slave programs, where one active object is responsible for placing data in the object space, while multiple slave active objects retrieve the data and do any necessary processing on it.

The canonical example for demonstrating coordination using a tuple/object space is the dining philosophers problem. There are many variations of this problem, with varying degrees of detail. The example used here is the same as the one given in [16], which consists of philosophers, chopsticks, chairs, a door, and a waiter. The waiter, shown in Figure 5.9, is responsible for opening/closing the restaurant, setting the table, and cleaning the dirty chopsticks. For the purpose of this example, the waiter is also responsible for creating the philosophers.

The closing of the restaurant is indicated by the removal of the `Door` object from the object space by the waiter. A new door, created in the "closed" state is then placed back into the object space so that the philosophers can't enter the restaurant (the door is closed). This is an ideal case of where the `rmOut()` function, introduced in Section 4.1.4

```
import clientside.ObjectSpaceJobTemplate;
import gridutil.objectspace.*;

/** This class is the waiter in the classical Dining Philosophers problem. The waiter is responsible
    for opening the restaraunt, setting the table, cleaning the table, and closing the restaraunt. */
public class Waiter extends ObjectSpaceJobTemplate
{
    /** The number of seats in the restaurant (and chopsticks). */
    private final int NUM_SEATS = 5;

    /** The number of philosophers for the program. */
    private final int NUM_PHIL = 3;

    /** How long the restaurant will be open for (in milliseconds). */
    private final long OPEN_HOURS = 30000; // 30 seconds

    /** The waiter first lays out the chopsticks and seats. Next the waiter creates all the philosophers,
        and then open the store. While the store is open, the waiter cleans the chopsticks. When the
        restaurant closes, the waiter put away the seats and chopsticks. */
    public void execute()
    {
        long openingTime;
        MultiSet m = new MultiSet();

        // layout chopsticks on table
        for(int i = 0; i < NUM_SEATS; i++)
        {
            m.put(new Seat(i));
            m.put(new Chopstick(i, (i+1) % NUM_SEATS));
        }
        self.out(m, ObjectSpace.INFINITE_TIMEOUT);

        // create philosophers
        m = new MultiSet();
        for(int i = 0; i < NUM_PHIL; i++)
            m.put(new Philosopher());
        self.eval(m, ObjectSpace.INFINITE_TIMEOUT);

        // open for business
        openingTime = System.currentTimeMillis();
        m = new MultiSet();
        m.put(new Door(true));
        self.out(m, ObjectSpace.INFINITE_TIMEOUT);

        while(System.currentTimeMillis() - openingTime < OPEN_HOURS)
            cleanChopsticks();

        // close the door
        m = self.in(new Door(), 1, 1, ObjectSpace.INFINITE_TIMEOUT);
        m = new MultiSet();
        m.put(new Door(false));
        self.out(m, ObjectSpace.INFINITE_TIMEOUT);

        // clean up objectspace
        self.rm(new Seat(), NUM_SEATS, NUM_SEATS, ObjectSpace.INFINITE_TIMEOUT);
        cleanChopsticks();
        for(int i = 0; i < NUM_SEATS; i++)
            self.rm(new Chopstick(i), 1, 2, 1000);
    }
```

Figure 5.9: The waiter for the Dining Philosophers problem (part 1)

```
    /** While the restaurant is open, the waiter periodically needs to clean the dirty chopsticks off
        the table and replace them with clean ones. */
    private void cleanChopsticks()
    {
        Chopstick c = new Chopstick(0);
        c.setMatchDirty();

        MultiSet dirty = self.in(c, 0, NUM_SEATS, 5000); // 5 seconds
        if(dirty.numItems() != 0)
        {
            MultiSet clean = new MultiSet();
            for(int i = 0; i < dirty.numItems(); i++)
            {
                c = (Chopstick) dirty.get(i);
                c.markClean();
                clean.put(c);
            }
            self.out(clean, ObjectSpace.INFINITE_TIMEOUT);
        }
    }
}
```

Figure 5.9: The waiter for the Dining Philosophers problem (part 2)

should be used. Because the door removed by the waiter is not really important, and is immediately replaced by a different object, the `rmOut()` function could be used instead. Thus, the code to close the restaurant can be changed to:

```
m = new MultiSet();
m.put(new Door(false));
self.rmOut(new Door(), 1, 1, ObjectSpace.INFINITE_TIMEOUT, m);
```

Hence, from the philosopher's point of view, there will *always* be an instance of `Door` in the object space, because the removal of the open door and replacement by a closed door is an atomic operation.

Before examining the philosopher, it is important to realize that the true power of this program hinges on how chopsticks are obtained. Finding matching objects in the object space is done using the `match()` method. However, the waiter wants to find dirty chopsticks, while the philosopher is interested only in the chopsticks on either side of their seat. Thus, it is important to realize how matching chopsticks is done. Figure 5.10 shows the fields and method from the `Chopstick` class that are essential to performing matches. Recall that the `match()` method is called on the template object passed to the object space, *not* the objects already in the object space. This allows the waiter to pass in a dirty chopstick to the object space to find all the dirty chopsticks. The philosopher passes in a chopstick with his current seat number, so the chopsticks on either side of the seat can be found.

The philosopher, shown in Figure 5.11, tries to enter the restaurant if the door is open.

```
/** Flags for dealing with a dirty/used chopstick. */
private boolean isDirty, matchIfDirty;

/** The various seat positions related to this chopstick. */
private int leftSeat, rightSeat, mySeat;

/** How this method operates depends on whether or not it is matching dirty chopsticks. If we want
    to match dirty ones (only done by Waiter), all dirty chopsticks match, regardless of their seat.
    If we want clean chopsticks, they only match if they are to the left or right of the seat
    associated with 'this' chopstick. */
public boolean match(OILObject obj)
{
    if(obj instanceof Chopstick)
    {
        Chopstick temp = (Chopstick) obj;
        if(matchIfDirty) // find a dirty chopstick
            return temp.isDirty;
        // find the chopstick on the other side of the plate
        return !temp.isDirty && (temp.leftSeat == mySeat || temp.rightSeat == mySeat);
    }
    return false;
}
```

Figure 5.10: Part of the `Chopstick` class

He then tries to find a free seat, and the chopsticks on either side of the seat. Since the philosophers are busy people, they are only willing to wait so long for both seats and chopsticks. If neither can be obtained, the philosophers leave the restaurant to go think. After they're done thinking, the whole process repeats. For the purpose of this example, when the restaurant is closed, the philosophers place themselves in the object space then finish executing. By placing themselves in the object, when the result for the program are displayed, it is possible to see how many times each philosopher managed to get something to eat. The `eat()` and `think()` methods are not shown.

5.5.3 Semaphores

The semaphore programming model can easily be emulated with the object space. Each critical section that is protected by a different semaphore can have a different semaphore class protecting it. For example, for the standard multiple readers, single writer problem the following could be used:

```
MultiSet multi = new MultiSet();
WriterSemaphore wSem = new WriterSemaphore();
multi.put(wSem);
for(int i = 0; i < 5; i++)
{
    ReaderSemaphore rSem = new ReaderSemaphore();
    multi.put(rSem);
}
self.out(multi, ObjectSpace.INFINITE_TIMEOUT);
```

```
import clientside.ObjectSpaceJobTemplate;
import gridutil.objectspace.*;

import java.util.Random;

public class Philosopher extends ObjectSpaceJobTemplate
{
    /** A counter for the number of times this Philosopher managed to eat. */
    private int timesEaten;

    /** A random number generator, used for creating random time lengths for eating and
        thinking. */
    private Random rand = new Random();

    /** The Philosopher first checks to make sure the restaurant is open. He then tries to get a
        free seat. Once he is seated, he tries and get the two chopsticks for his seat. If he
        gets them he can eat, and then places the dirty chopsticks and his seat back in the
        object space. Finally, he goes to think for a while and the loop repeats. */
    public void execute()
    {
        Chopstick c1, c2;
        MultiSet m, table;
        Door d = new Door();
        Seat s = new Seat();

        self = null;
        m = context.rd(d, 1, 1, ObjectSpace.INFINITE_TIMEOUT);
        d = (Door) m.get(0);
        while(d.open)
        {
            m = context.in(s, 1, 1, 10000); // wait max 10 seconds for a free seat
            if(m.numItems() == 1)
            {
                s = (Seat) m.get(0);
                c1 = new Chopstick(s.number);
                table = new MultiSet();

                // try and grab two chopsticks
                m = context.in(c1, 2, 2, 5000); // wait max 5 seconds for chopsticks

                // have both chopsticks
                if(m.numItems() == 2)
                {
                    c1 = (Chopstick) m.get(0);
                    c2 = (Chopstick) m.get(1);
                    eat();
                    c1.markDirty();
                    c2.markDirty();

                    // place chopsticks back on table
                    table.put(c1);
                    table.put(c2);
                }
                table.put(s); // get out of seat
                context.out(table, ObjectSpace.INFINITE_TIMEOUT);
            }
            think(); // leave restaurant to think
            m = context.rd(d, 1, 1, ObjectSpace.INFINITE_TIMEOUT);
            d = (Door) m.get(0);
        }

        // used to collect results, outside of main problem area
        m = new MultiSet();
        m.put(this);
        context.out(m, ObjectSpace.INFINITE_TIMEOUT);
    }
```

Figure 5.11: The philosopher for the Dining Philosophers problem (partial listing)

Thus, when an active object needs to enter the read critical section (maximum five allowed entrance at a time), it merely removes the appropriate semaphore from the object space. When the critical section is exited, the semaphore can simply be placed back into the object space. For the writer to enter its critical section, it obtains the proper semaphore (`wSem`), and will also need to read all the instances of `ReadSemaphore` (`rSem`) from the object space to make sure no readers are active while writing is taking place. Once it has exited its critical section, both the `wSem` and `rSem` will need to be placed back into the object space.

5.5.4 Message Passing

One of the main strengths of message-passing libraries, such as PVM and MPI, is the automatic translation of data types between heterogeneous machines. This feature is really a non-issue in JOLTS because of the fact it is based on Java, translation between different architectures is handled by the JVM. There are two main types of message passing, unicast and multicast, both of which can be emulated using an object space.

Unicast message passing uses the sending/receiving of the message as a synchronization point, where both the sender and receiver can only advance once they have *both* reached their half of the message. The receiver is simpler because it can check the object space for the message, and whether or not the message is present indicates whether or not the sender is at the synchronization point. The sender, on the other hand, is more complicated. When the message is placed in the object space, there is no guarantee that the receiver is ready for the message. As a result, the sender must wait for some form of notification that the receiver has in fact received the sent message. The are two options for this problem. The first option is to indicate message reception by the absence of the sent message in the object space. The second option requires the receiver to place some type of acknowledgement message back into the object space. The messages for either option can be defined as follows:

```
class Message implements OILObject
{
    public final String sender, receiver;
    public final Object payload;

    public boolean match(OILObject obj)
    {
        if(obj instanceof Message)
        {
            Message temp = (Message) obj;
            return sender.equals(temp.sender) && receiver.equals(temp.receiver);
        }
        return false;
    }

    ⋮  // any additional methods
}
```

When the sender creates the message, it need to properly identify itself as the sender, and the ID of the receiver. To receive the message, the receiver needs to know the ID of the sender so the proper `Message` can be retrieved from the object space. This type of class is very close to the message-passing idea that both the sender and receiver know about each other. JOLTS has the advantage of removing the requirement of knowing what computer the sender/receiver is one, only some type of unique ID is required; e.g., a `String` in the above `Message`.

Emulating multicast message-passing is very similar. Since there are multiple receivers with multicast, the problem of the sender recognizing that all the receivers got the proper message is a little more of a challenge. The challenge is whether or not the receivers are allowed to continue executing as they got the message, or only after *all* the receivers have received the message. If the former case is desired, the receivers can simply place an acknowledgement message back in the object space and continue executing. If the later case is required, each receiver will need to place an acknowledgement message in the object space, and then check for the existence of the proper number of acknowledgements in the object space; i.e., the number of receivers. Once the proper number is seen, all the receivers can continue executing. If desired, the sender can then remove the acknowledgements from the object space.

Chapter 6

Conclusion

This chapter is meant to just be a short summary of both the research contributions of this thesis, and the future related work in Sections 6.1 and 6.2, respectively.

6.1 Research Contributions

The research contribution of this thesis can broadly be grouped into three main areas:

1. The JOLTS system.
2. The `rm` and `rmOut` additions to Objective Linda.
3. The Timed Function Execution Pattern.

Each of these areas are discussed in turn in the following subsections.

6.1.1 JOLTS

Many of the papers that discuss Linda [4, 18], and its subsequent extensions [10, 15, 16, 18, 20], have few, if any, performance metrics or discussion of how to implement this coordination model. In many of the papers it is even unclear whether the system being discussed in run locally, or distributed across multiple machines. While a large part of this thesis is the implementation of the JOLTS system, it is meant not only to be a usable grid system, but a design document describing how a particular extension of Linda, the Objective Linda extension, can be implemented.

A large amount of research has gone into the Linda coordination model as seen in Section 2.2. Extensions usually consist of modifying the primitives, and adding new primitives. No system appears to have a pervasive checkpoint mechanism such as the one

offered by JOLTS. As mentioned in the previous paragraph, it is unclear if the other systems designed around Linda and its extensions use a single, or multiple machines. If they use only a single machine, the lack of a checkpoint mechanism is understandable. However, when multiple machines are being used, such as in JOLTS, checkpointing becomes a very useful feature. Without checkpoints, any program, whether or not it is run on a grid, needs to be restarted from the beginning.

Chapter 5 gave experimental results of the JOLTS system. It shows that this particular implementation of object spaces is a viable form of coordination in a grid system. The server hosting the object space doesn't need to be a powerful computer, merely one that has a fast network connection. The processing power of the server is almost of secondary importance. When checkpoints are used, the performance hit is negligible on a fast network. If the checkpoints are rather large, checkpointing *will* have an impact on performance; however, it is felt that any performance penalty is more than offset by the protection offered against worker-node failure.

It is important to realize that while the JOLTS system is designed to handle object space jobs, the system didn't initially support such a complex job structure. Section 5.5 gives examples of some of the other programming models supported by JOLTS. For a more thorough discussion of additional example programs for JOLTS, as well as to obtain the latest build of the system, refer to [19].

6.1.2 Extensions to Objective Linda

This thesis proposes two new primitives to the Objective Linda specification: `rm` and `rmOut`. These two primitives are really only important to implementations that operate over a network utilizing multiple computers. Without these two new primitives, any time an object needs to be deleted from an object space using the `in` primitive, valuable network bandwidth is wasted transmitting the results to an active object that doesn't require the results. The `rm` primitive is meant to eliminate that wasted bandwidth by only transmitting the number of objects removed from the object space. This value is small enough to fit inside a single network packet, keeping the network usage to an absolute minimum.

A feature that appears to be missing from Linda and all of its various extensions is the ability to modify an object already in an object space. Because the object space is a

shared-memory area, it makes sense to protect it from potentially deadly modifications. Thus, the only way to change an existing object was to remove the object, and then replace it with a modified form. Because these are two separate operations, it is possible that the time between the removal and the insertion of a replacement can cause problems for other active objects using the object space. The solution is the `rmOut` primitive introduced in this thesis. It combines the tasks of removing an object and replacing it with another object into a single atomic operation. There is no longer a visible gap to another active object where the object in question isn't present in the object space.

Neither of these two new primitives are *necessary* for JOLTS to work; however, their addition does help eliminate network waste. It is important to eliminate all unnecessary network traffic because, as shown in Section 5.4, network bandwidth can very quickly become the limiting factor of job performance in JOLTS.

6.1.3 Timed Function Execution Pattern

Using patterns when designing and building large systems is a growing trend. Patterns are an efficient way to document common problem/solution pairs that can then be used by other software developers when the same problem arises in a different project. This thesis documents a new design pattern called the *Timed Function Execution Pattern*. The word "timed" in this case does not mean *examining* how long a function is to execute, but instead refers to *limiting* the amount of time a function has to execute.

Many of the primitives offered by Linda and its extensions have timeout values. None of the papers that discuss Linda or any of its various extensions describe how these timed functions are implemented. To deal with this special class of functions, the Timed Function Execution Pattern, described in Section 4.3.3.1, was created. There are two possible ways to implement this design pattern, depending on the situation where it will be used. Option one uses a multi-threaded approach, while option two uses a loop checking mechanism described in Appendices A.2 and A.3.2, respectively.

While this pattern may not be as widely applicable as some patterns, such as the Facade pattern in information systems, it is indispensable when its problem/solution pair arises in a system. It obviously can be used by anybody implementing a system based on Linda, but its use isn't limited just to grid systems. For example, it can be used to limit the amount of time a genetic algorithm executes.

6.2 Future Work

Like any medium-scale system, there is always additional features that could be added, and JOLTS is no different. One major assumption when creating JOLTS was that the main server wouldn't fail. For very long running jobs, on the order of several months, it may be required to shutdown/restart the server for a variety of reasons. An additional feature would be to give the server itself the ability to checkpoint, so when it is restarted any jobs that were executing when the server went down would automatically restart. There are several different ways this feature could potentially be implemented. Further research will be required to determine which method is most desirable. Additionally, resuming the server from a checkpoint would also require the ability to checkpoint entire object-space jobs.

As noted in Section 4.3.3, it is not possible to checkpoint an object space, only the active objects that manipulate the object space. This is because there is no way to guarantee that every active object is in a safe state. Such a feature is desirable, but determining how to design and implement such capabilities is a non-trivial task. Also, there are many unanswered questions regarding such a feature that include:

- Should a polling mechanism be used between all the active objects to allow checkpointing, or can any active object cause an object space to be checkpointed?
- Is only one object space checkpointed, or are *all* the object spaces associated with the user's job?
- How do you recover an object space from a saved checkpoint?
- How do you inform currently executing active objects that an object space they are using is about to be recovered from a checkpoint, and what is the checkpoint number?

All of these questions need to be answered before such a feature could possibly be incorporated into JOLTS. This feature, along with the previously mentioned server checkpointing would add a great deal of flexibility to JOLTS.

One of the more interesting features of JOLTS, from an implementation point of view, is the security manager. It offers great flexibility allowing the administrator to configure

how open or restrictive the JOLTS server behaves. However, as noted in Appendix A.1, the list of restricted methods is currently hardcoded into the system. To increase the ease of security configuration, a graphical interface could be implemented that allowed the administrator more control over what components of the Java library are restricted, *including* methods. This tool would be responsible for translating the list of restricted components between human-readable and JVM-readable forms.

While both the JOLTS server and worker components can be configured using a properties file, these components are rather static once they are started. There is no way to dynamically update any of the configuration values once the component is running. This is desirable for the server, but not necessarily for the worker nodes. A feature that *could* be added to the worker node is the ability to dynamically change some of the configuration values while the node is running. For example, some person using their desktop PC as a worker node may allow JOLTS full CPU control during the night when the computer is not being used, but may only want to give 40% of their CPU to JOLTS during the day while they are working. To do this now, the worker node would need to be shut down, the configuration file changed, and the worker node restarted. Ideally the file could be changed, or a graphical interface used, to dynamically update the worker node with how much CPU the user is willing to dedicate to JOLTS at the current time. This feature is not necessarily very important, it is more of a convenience than anything else.

Java was picked as the implementation language for JOLTS for several reasons, including the sandbox environment created by the JVM, and the fact that the implementor was very familiar with the language. It has been suggested that JOLTS should be ported over to the .Net platform. This will open up the JOLTS system to more potential languages. The system is well documented, as shown throughout Chapter 4, so porting JOLTS to another language shouldn't be too complicated, with one exception, the security subsystem. The security mechanism in JOLTS is *highly* dependent on the capabilities of the JVM, and how Java bytecode is defined. The .Net platform has an equivalent to Java bytecode, so porting *should* be possible; however, the security system will most likely require a complete rewrite. Whether the porting of JOLTS to .Net takes place is undetermined, mainly because the size of such an undertaking would require several months of full-time effort by a highly skilled programmer, knowledgeable in .Net and all it's capabilities and limitations.

References

[1] Group 3. Encryption. http://www.cs.usask.ca/classes/371/projects/99group3/, June 2004.

[2] Victor R. Basili and H. Dieter Rombach. The TAME Project: Towards Improvement Oriented Software Environment. *IEEE Transactions on Software Engineering*, 14(6), pages 758–773, June 1988.

[3] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Toronto, Canada: Addison-Wesley, 1st edition, 1998.

[4] Nicholas Carriero and David Gelernter. Linda in context. *Commun. ACM*, 32(4), pages 444–458, 1989.

[5] MPI Forum. Message passing interface (mpi) forum home page. http://www.mpi-forum.org/, January 2004.

[6] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. San Francisco, CA: Morgan Kaufmann Publishers, Inc., 1st edition, 1999.

[7] Ian Foster, Carl Kesselman, Jeffery Nick, and Steven Tuecke. The Physiology of the Grid – An Open Grid Services Architecture for Distributed Systems Integration. In Fran Berman, Geooffery Fox, and Anthony Hey, editors, *Grid Computing – Making the Global Infrastructure a Reality*, chapter 8. West Sussex, England: John Wiley & Sons Ltd., 2003.

[8] Dennis Gannon, Randy Bramley, Thomas Stuckey, Juan Villacís, Jayashree Balasubramanian, Esra Akman, Fabian Breg, Shridar Diwan, and Madhu Govindaraju. Component Architectures for Distributed Scientific Problem Solving. *IEEE Computational Science and Engineering*, 5(2), pages 50–63, 1998.

[9] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing*. Cambridge, MA: The MIT Press, 1994.

[10] David Gelernter and David Kaminsky. Supercomputing Out of Recycled Garbage: Preliminary Experience with Piranha. In *6th ACM International Conference on Supercomputing*, pages 417–427, Washington, D.C., 1992.

[11] William Gropp and Ewing Lusk. PVM and MPI Are Completely Different. Technical report, Argonne National Laboratory, September 1998.

[12] Ron Hitchens. *Java NIO*. Sebastopol, CA: O'Reilly & Associates, Inc., 2001.

[13] Innovative Computing Laboratory. Netsolve. http://icl.cs.utk.edu/netsolve/, January 2004.

[14] Leander Kahney. Cheaters Bow to Peer Pressure. http://www.wired.com/news/infostructure/0,1377,41838,00.html, February 2001.

[15] Thilo Kielmann. Object-Oriented Distributed Programming with Objective Linda. In *First International Workshop on High Speed Networks and Open distributed Platforms*, June 1995.

[16] Thilo Kielmann. Designing a Coordination Model for Open Systems. In P. Ciancarini and C. Hankin, editors, *Proc. 1st Int. Conf. on Coordination Models and Languages*, volume 1061, pages 267–284, Cesena, Italy, 1996. Springer-Verlag, Berlin.

[17] LinuxHQ. Shared memory. http://www.linuxhq.com/guides/LPG/node65.html, February 2004.

[18] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. Technical report, Centrum voor Wiskunde en Informatica (CWI), 1998.

[19] Jeremy Pfeifer. JOLTS Homepage. http://www.cs.usask.ca/~jpp960/thesis/jolts.php, July 2004.

[20] Antony Rowstron and Alan Wood. Bonita: A set of tuple space primitives for distributed coordination. In *Proc. HICSS30, Sw Track*, pages 379–388, Hawaii, 1997. IEEE Computer Society Press.

[21] Douglas Schmidt, Michael Stal, Hans Rohnert, and Fank Buschmann. *Pattern-Oriented Software Architecture – Patterns for Concurrent and Networked Objects*, volume 2. Toronto, Canada: John Wiley & Sons, 2000.

[22] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI – The Complete Reference: Volume 1, The MPI Core*. Cambridge, MA: The MIT Press, 2nd edition, 1996.

[23] Niranjan Suri, Jeffrey M. Bradshaw, Maggie R. Breedy, Paul T. Groth, Gregory A. Hill, Renia Jeffers, and Timothy S. Mitrovich. An overview of the nomads mobile agent system. In *Proceedings of ECOOP'2000*, Nice, France, 2000.

[24] Condor Team. *Condor Version 6.6.0 Manual*. University of Wisconsin-Madison, January 2004.

[25] Top 500 Supercomputer Sites. Top 500 list. http://www.top500.org/list/2003/11/, December 2003.

[26] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A High-Performance Java Dialect. In *ACM 1998 Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.

Appendix A

Implementation Details

While Chapter 4 described the high-level functionality of the JOLTS system, as well as *what* some of the various classes do, very few details were given as to *how* these classes worked. This appendix is meant to provide detailed discussion on how some the more interesting aspects of the JOLTS system are actually implemented. The main areas that will be discussed include:

- how the security manager works,
- implementing the Time Function Execution pattern,
- implementing the object space for both the worker and server, and
- dealing with a multiplexor.

It is assumed that the reader has a detailed understanding of the features in the Java language, and a *basic* understanding of how the Java Virtual Machine (JVM) works. For the discussions relevant to the Timed Function Execution pattern, a basic understanding of threads and critical sections is also needed.

A.1 JOLTS Security Manager

The original goal of the security manager was to make sure the class indicated by the user as the starting class for the submitted job was present in the submitted class/jar file, and that is was a descendant of a valid interface defined in the `clientside` package. Over time the security manager evolved into a class capable of enforcing restrictions that aren't normally possible in Java, such as making sure a specific method is never called in a class. The main features of the security manager include:

1. Making sure restricted classes aren't used.
2. Making sure restricted fields aren't accessed.
3. Making sure restricted methods aren't invoked.
4. Ensuring the proper constructor exists for SIMD sub-jobs.
5. Ensure the checkpoint mechanism (if used) is declared as **`transient`**.
6. Allowing only authorized clients to submit jobs.

The first three features are only capable with access to the class files for the job being submitted. This obviously isn't a problem as the user must submit his compiled code to the JOLTS system for the system to know what code to run.

The first step in performing the first three security checks involves processing the class *file* looking for the relevant data. Simply loading the class and using reflection is unable to handle the first three security checks (although it is used for checks 4 and 5). Often, examining certain properties using reflection will raise security exceptions in the JVM; however, by examining the class files directly (in byte form) no such problem exists. The *entire* class file doesn't need to be checked, the only area of interest is the *constant pool*. Before the discussion can proceed, a basic understanding of the structure of the Java class file, specifically the constant pool, is required.

The Java class file is a binary file. All information stored inside the file is based on unsigned 8-byte characters (with a few exceptions). Multiple bytes can be grouped together to form 2-byte or 4-byte numbers. Bytes are always written to the class file, high-order byte first. The first eight bytes in the class file are the numbers 0xCAFEBABE (in hexadecimal), 3, and 45, which are the magic, Java minor, and Java major version numbers, respectively. The next two bytes indicate the number of entries in the constant pool. In a Java program, any time something other than a local variable or parameter is being used, chances are it is being accessed from the constant pool. The constant pool is used to store several different types of information.

- Literals
  - all string literals
  - float literals (other than 0.0, 1.0, and 2.0)
  - integer literals greater than 16-bits in size
- Method names and signatures
- Class variables
- Instance variables
- Other class names

The constant pool is a large byte array, where consecutive entries combine to create simple data types. Constant pool entries can vary in length. This makes jumping to a particular entry difficult, since all previous entries must be checked first to determine how large they are, so the start of the following entry can be found. The first byte inside each constant pool entry indicates what type of structure it is. Each type is represented in the hierarchy given in Figure A.1. These names are short forms for the names used inside the JVM. The structures relevant to the security manager will now be discussed in turn.

- UTF8 — The most common constant pool structure is the UTF8. UTF8 is an encoding scheme for string values, designed to allow ASCII-based strings to be represented using a single byte, but still retain the functionality required to also represent unicode characters. Each UTF8 structure consists of $3 + n$ bytes, where n is the length of the string being represented. The first byte is 1, indicating that it is a UTF8 structure. The next two bytes indicate the length of the string, and the remaining n bytes contain the string (in ASCII form).

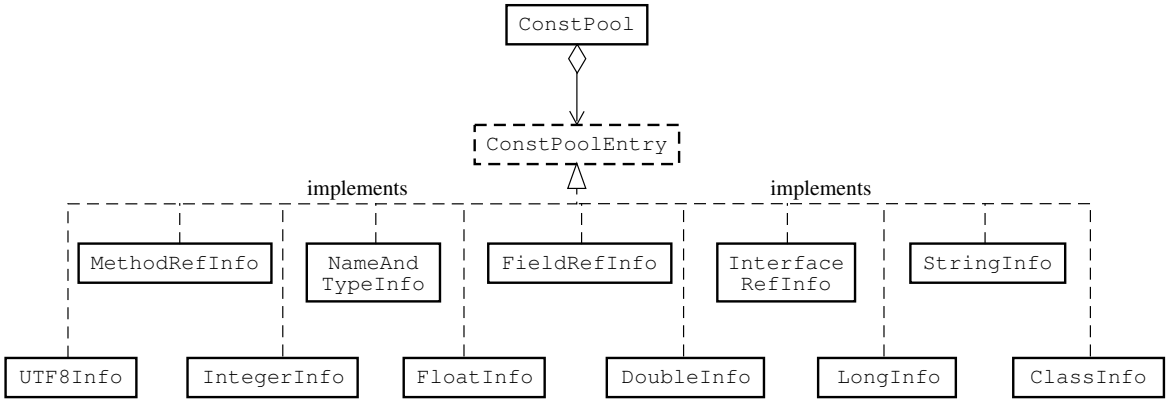

Figure A.1: The `gridutil.constantpool` package

- Class — The Class structure contains three bytes. The first byte is a 7, to represent that the structure is for a Class. The other two bytes are used as an index into the constant pool. At the specified index a UTF8 must be found, containing the name of the class in fully qualified form.

  > **Note:** the fully qualified form for a class name inside the virtual machine is slightly different than that used in the Java language. Where a fully qualified name usually has periods as separators; e.g, `java.lang.String`, the periods are replaced with forward slashes inside the virtual machine; e.g., `java/lang/String`.

- Fieldref—The Fieldref structure contains five bytes. The first byte is a 9, to represent that the structure is for a Fieldref. The next two bytes are used as an index into the constant pool. At the specified index, a Class structure must be found, which is the class that contains the field. The final two bytes are also used as an index into the constant pool. At the specified index, a NameAndType structure must be found, which represents the name and type of the field.

- Methodref — The Methodref structure contains five bytes. The first byte is a 10, to represent that the structure is for a Methodref. The next two bytes are used as an index into the constant pool. At the specified index, a Class structure must be found, which is the class that contains the method. The final two bytes are also used as an index into the constant pool. At the specified index, a NameAndType structure must be found, which represents the name and signature of the method.

- NameAndType — The NameAndType structure contains five bytes. The first byte is 12, to represent that the structure is for a NameAndType. The next two bytes are used as an index into the constant pool. At the specified index, a UTF8 structure must be found, which is the "name." The final two bytes are also used as an index into the constant pool. At the specified index, a UTF8 structure must be found, which represents the "type" or descriptor associated with the name. This structure never exists on its own, it is also pointed to by another structure. This other structure is what indicates what the "name" and "type" are supposed to represent.

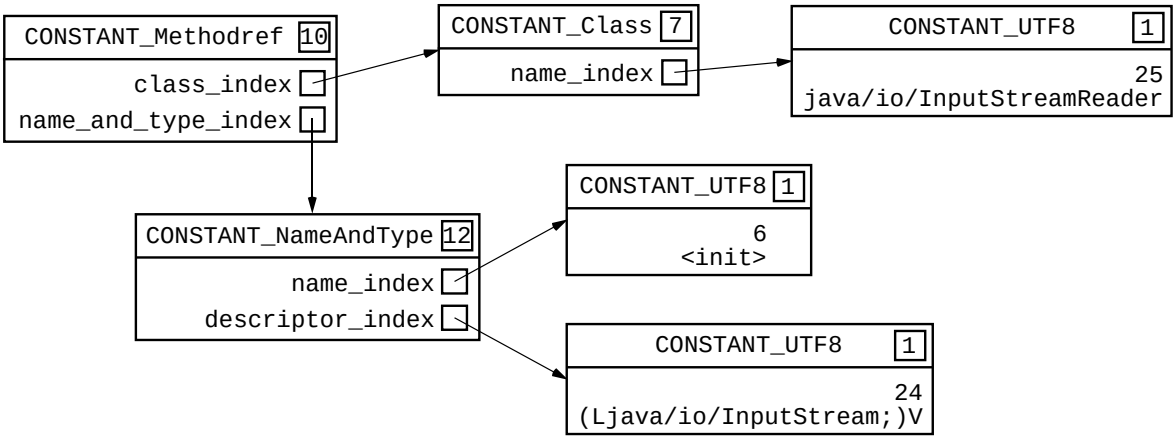

Figure A.2: Methodref structure for `InputStreamReader` constructor

To better understand these various constant pool structures, consider a simple example. When creating a keyboard reader in Java, a `BufferedReader` must be attached to `System.in`. This is usually accomplished in a line of code such as

```
BufferedReader keyboard = new BufferedReader(new InputStreamReader(System.in));
```

The first step is to create a `BufferedReader` object and place it on the operand stack. The second step is to create an `InputStreamReader` object and place it on the operand stack. The third step in executing this statement is to place the field `System.in` on the top of the operand stack. The fourth step is to call the constructor for the `InputStreamReader` class, using the field `System.in` as the argument to the constructor. This is done using opcode 183 (`invokespecial`), followed by the location of a Methodref that represents this constructor. This Methodref structure is given in Figure A.2. Notice that the Methodref structure has a value 10, indicating that the structure is in fact a Methodref. Its `class_index` points to a Class, which points to a UTF8 containing the name of the class. The NameAndType is used to describe the signature of the method. The special name `<init>` indicates that it is a constructor, while the `descriptor_index` points to the constructor signature.

As demonstrated in the previous example, determining if methods are used from other classes can be accomplished by looking at the constant pool. The `ConstPool` class is designed to take the raw bytes from the constant pool of a class file and turn them into a more usable form for traversal. The class is designed to create an array of `ConstPoolEntry` where the index of each entry correspond to the position of the entry in the byte-form constant pool from the class file. It is much easier to search through the entries for a specific entry in this form, than to be jumping around in a byte array. For example, by default, user programs aren't allowed to use the `FileReader` class. This is because the program could then read files from the worker node's disk, clearly a potential security problem. Thus, if any class that is submitted with the user's job tries to use the `FileReader` class it will appear in the class's constant pool, detected by the security manager, and the job will be rejected. The code to perform the search is:

```
public boolean usesClass(String str)
{
    str = str.replace('.', '/'); // convert to proper JVM format
    for(int i = 1; i < entries.length; i++)
        if(entries[i] instanceof ClassInfo)
        {
            ClassInfo temp = (ClassInfo) entries[i];
            int nameIndex = temp.getIndex();
            UTF8Info temp2 = (UTF8Info) entries[nameIndex];
            if(temp2.getUTF8().equals(str)) // found matching String name
                return true;
        }
        else if(entries[i] instanceof NameAndTypeInfo)
        {
            NameAndTypeInfo temp = (NameAndTypeInfo) entries[i];
            int descripIndex = temp.getDescripIndex();
            UTF8Info temp2 = (UTF8Info) entries[descripIndex];

            /*  Class names in parameter list and return types are modified slightly, so
                this modification makes it match the JVM spec. */
            if(temp2.getUTF8().indexOf("L" + str + ";") != -1)
                return true;
        }
    return false;
}
```

This method checks for both direct classes and parameters types in method signatures. To check if a class uses `FileReader`, the fully qualified name is passed in as an argument:

```
boolean uses = constPool.usesClass("java.io.FileReader");
```

This process is repeated for each forbidden class, and is run on each class that is part of the submitted job. A similar process is used for checking for restricted fields and methods.

The list of restricted classes and fields is contained in a configuration file read by the server when it starts up. This allows the administrator of the system to create a more or less secure grid to run programs in. The restricted *methods* are currently hardcoded in the `GridSecurityManager` class. It would be possible to allow these to also be set in a configuration file; however, the format required is based on the internal form of the JVM, which most administrators don't know. Thus, to prevent more harm than good, the list of restricted methods can't be modified.

The last three functions of the security manager, listed on page 127, can be performed using standard reflection techniques (features 4 and 5), and network programming (feature 6). Making sure any checkpoint fields are **`transient`** is rather simple:

```
/** This class is responsible for checking a class to see if it has any non-transient fields that
    are of type {@link CheckpointMech}. If one is found, an exception is thrown with a message
    giving the name of the class and the field in question.
    @param c The class to be checked.
    @throws GridException Thrown if a non-transient {@link CheckpointMech} field is found in the
        parameter class. */
private void checkCheckpointMechField(Class c) throws GridException
{
    Field fields[ ] = c.getDeclaredFields();
    for(int i = 0; i < fields.length; i++)
    {
        Class type = fields[i].getType();
        if(type.getName().equals("gridutil.CheckpointMech"))
        {
            int modifiers = fields[i].getModifiers();
            if(!Modifier.isTransient(modifiers))
                throw new GridException("Class '" + c.getName()
                        + "' has a non-transient checkpoint mech field '"
                        + fields[i].getName() + "'.");
        }
    }
}
```

A similar reflection technique is used to check the constructor of SIMD sub-jobs

```
result = loader.loadClass(jobName);
Constructor cons[ ] = result.getDeclaredConstructors();
for(int i = 0; i < cons.length; i++)
{
    params = cons[i].getParameterTypes();
    if(params.length != 1)
        continue;
    // constructor only has 1 parameter of type clientside.Parameters
    if(params[0].getName().equals("clientside.Parameters"))
        return simd;
}
throw new GridException("Class '" + jobName
        + "' doesn't have a constructor with the required parameters.");
```

The check for whether or not a client is authorized to submit jobs is simply a matter of checking the IP address of the client. Under the current JOLTS configuration, only clients from the University of Saskatchewan are allowed to submit jobs.

A.2 Implementing the Worker Node Object Space

As mentioned in Section 4.3.3.1, there are two ways to implement the Timed Function Execution pattern, each of which is used in the JOLTS system. This section describes the multi-threaded approach used on the worker nodes, while Appendix A.3.2 discusses the loop approach used on the server.

When object space jobs are executed on a worker node, the instance of `ObjectSpace` passed to the `setSelf()` and `setContext()` methods, shown in Figure A.3, are actually instances of `ObjectSpaceStub`. The important instance variable to note is `helper` inside class `ObjectSpaceStub`. Its type is declared as the abstract class `StubHelper`, whose hierarchy is given in Figure A.4. As can be seen by the name of the children in Figure A.4, each object space operation has a corresponding class. Some operations, such as `in` and `rd` are combined into a single class because the type and order of the data sent to the server, and the type of the returned result is the same. The important fact to note about

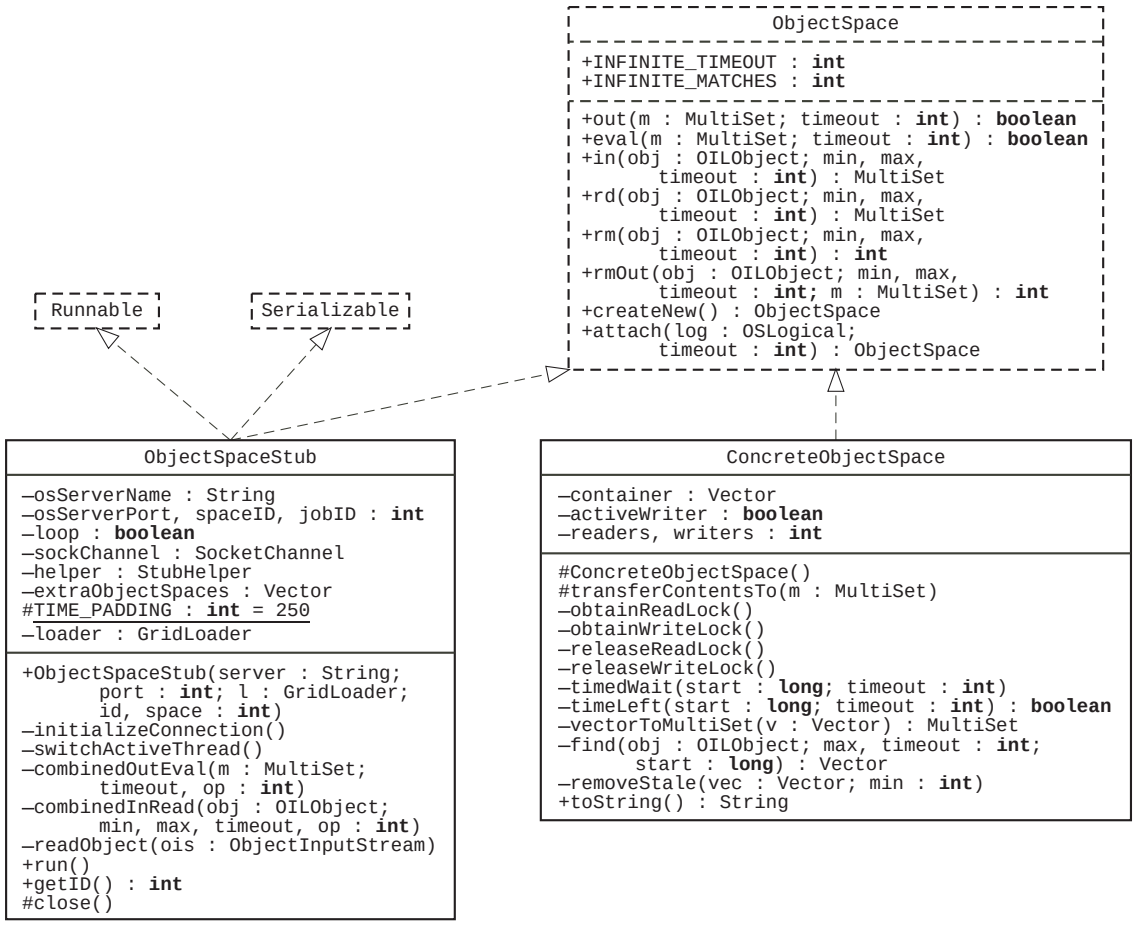

Figure A.3: The `ObjectSpace` hierarchy

class `ObjectSpaceStub` is that it implements interface `Runnable`.

When an instance of class `ObjectSpaceStub` is created by the system (user jobs can't *directly* create instances of this class), hidden inside the object is a thread, which is the "worker thread" in the Timed Function Execution pattern. This thread is responsible for executing the `helper` instance variable in the class `ObjectSpaceStub`. When an object space method is invoked, the method creates the appropriate helper object to do the network communication with the server, and then the primary thread goes into a wait state. The worker node then contacts the server (from inside the worker thread), sends the relevant information, and waits for the results. If the results arrive before the timeout expires, the results are placed in the `result` field in `StubHelper` and the worker thread wakes up the primary thread. Lastly, the worker thread goes back to sleep. The primary thread retrieves the result from the `helper` using `getResult()`, and typecasts it into the appropriate type. In the event the appropriate response was *not* received in time from the server, the value returned by method `getResult()` will be the default value for the requested object space method. The worker thread will *eventually* receive a response from the server, and which point the worker thread will go back to sleep. Consider the following example that demonstrates how the `rm()` method is performed.

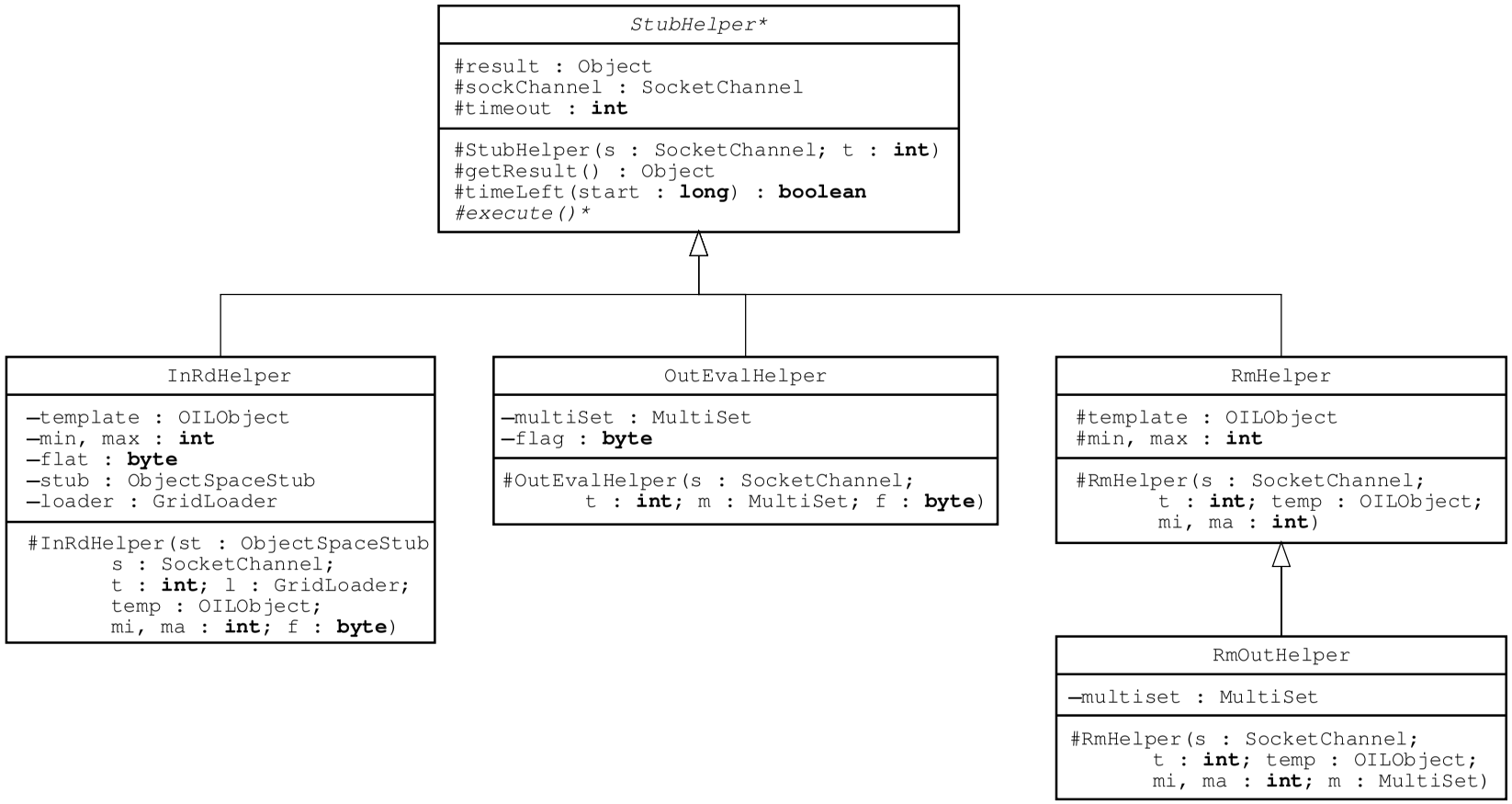

Figure A.4: The `StubHelper` hierarchy

```
public synchronized int rm(OILObject obj, int min, int max, int timeout)
        throws ObjectSpaceException
{
    verifyParameters(obj, min, max, timeout);
    helper = new RmHelper(sockChannel, timeout, obj, min, max);
    notify();
    try
    {
        // worker thread will now use newly created 'helper'
        wait(timeout + TIME_PADDING);
    }catch(InterruptedException e) {} // do nothing
    // time to return results from the helper object
    Integer temp = (Integer) helper.getResult();
    return temp.intValue();
}
```

The first line deals with enforcing the preconditions on object space methods. Once the parameters are checked, the `RmHelper` object is created. The call to method `notify()` causes the worker thread to wakeup; however, it won't actually start executing until the `wait()` method is called because it needs to get a lock on the object, which the primary thread currently has (note that this method is **`synchronized`**). Observe that a small amount of additional time padding is added to the wait length. This is used to account for a small amount of network delay. When the `wait()` returns, either by time expiring or by the worker thread waking it up, the result is retrieved from the **`helper`**, and the result is returned. Figure A.5 contains the complete source code for the `RmHelper` class, as a example helper class.

A similar structure is used for each of the remaining object-space primitives. The method in the stub class creates the appropriate helper object to communicate with the server in a different thread, while the primary thread waits for either a timeout or the response from the server.

Checkpointing causes and interesting problem for object-space jobs. Because communication with the real object space on the JOLTS server is done through socket channels, if an active object is resumed from a checkpoint, that socket channel will no longer be connected to the server. There are two possible solutions to this problem:

1. When an active object is resumed from a checkpoint, pass in fresh references to the object space(s) using the `setSelf()` and `setContext()` methods.

2. Have the underlying object space stub object automatically reconnect to the object space on the server itself.

Option 1 won't work for one simple reason: active objects can have additional object spaces beyond the default **`self`** and **`context`**. If new object space references need to be passed to the active object when it is resumed from a checkpoint, how would these references be passed to the active object? There is no way to ensure any additional object spaces are reconnected to the server by relying only on the two "set" methods. The object space must automatically reconnected itself to the server.

One of the important implementation details of class `ObjectSpaceStub` is the existence of a customized deserialization method, `readObject()`. When an object-space job is resumed from a checkpoint, as soon as the active object is deserialized on a worker node, the `ObjectSpaceStub` instance will attempt to re-attach itself to the server. Because any additional objects spaces created by the active object are in reality also instances of `ObjectSpaceStub`, they too will automatically reconnect to the server when the active object is deserialized.

```
package workerside.objectspace;

import gridutil.objectspace.OILObject;
import gridutil.MessageBytes;

import java.nio.channels.*;
import java.nio.ByteBuffer;
import java.io.*;

/** This class is responsible for sending the necessary data to the server to perform the {@link
    gridutil.objectspace.ObjectSpace#rm rm} operation on an object space. It also reads the result
    back, and makes it available to the calling class if the operation was performed in the
    required time.

    @author Jeremy Pfeifer
    @version 1.0 - initial implementation, April 7/04 */
public class RmHelper extends StubHelper
{
    /** A reference to the object that will be used as a template for performing the matches in the
        object space. */
    protected OILObject template;

    /** The minimum number of matches to find. */
    protected int min;

    /** The maximum number of matches to find. */
    protected int max;

    /** Simple constructor used to initialize the instance variables, both from this class and the
        ones inherited from the parent.
        @param s The channel attached to the object space on the server.
        @param t The timeout value of how long this operation is allowed to execute.
        @param temp The template used to find matches in the object space.
        @param mi The minimum number of matches to find.
        @param ma The maximum number of matches to find. */
    protected RmHelper(SocketChannel s, int t, OILObject temp, int mi, int ma)
    {
        super(s, t);
        template = temp;
        min = mi;
        max = ma;
        result = new Integer(0);
    }

    /** This method sends the relevant data (the arguments for the {@link
        gridutil.objectspace.ObjectSpace#rm rm} object space operation) to the server. It then reads
        the results of the operation from the server. If this all happened before the timeout
        expired, the results are stored in a location so that the {@link ObjectSpaceStub
        ObjectSpaceStub} can access them. Otherwise, the default value (<tt>0</tt>)) is left in the
        storage space.
        @throws IOException Thrown if a problem occurs while reading/writing to the channel. */
    protected void execute() throws IOException
    {
        int temp;
        long start = System.currentTimeMillis();
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(baos);
        oos.writeObject(template);
```

Figure A.5: The complete listing for `RmHelper` (part 1)

```
            byte array[ ] = baos.toByteArray();
            // sum all the bytes required
            ByteBuffer buffer = ByteBuffer.allocate(1 // the REMOVE flag
                + 4 // the min value
                + 4 // the max value
                + 4 // the timeout value
                + 4 // the template size
                + array.length); // the template data

            buffer.put(MessageBytes.REMOVE);
            buffer.putInt(min);
            buffer.putInt(max);
            buffer.putInt(timeout);
            buffer.putInt(array.length);
            buffer.put(array);
            buffer.flip();

            while(buffer.hasRemaining())
                sockChannel.write(buffer);

            buffer = ByteBuffer.allocate(4);
            try // in case timeout is reached and the job exists, closing the channel
            {
                sockChannel.read(buffer);
                buffer.flip();
                temp = buffer.getInt();

                if(timeLeft(start))
                    result = new Integer(temp);
            }catch(AsynchronousCloseException e)
            {
                System.err.println("Some other thread closed my socket while in Rm :(");
                sockChannel.close();
            }
        }
    }
```

Figure A.5: The complete listing for `RmHelper` (part 2)

On careful examination of Figure A.4, the reader will notice that class `InRdHelper` contains a `loader` field while none of the other classes in the hierarchy do. The reason is only the `in()` and `rd()` methods retrieve actual objects from the object space. Since these objects are coming from a socket channel, they will need to be deserialized upon their arrival. By default, the JVM uses the default class loader when deserializing objects. This will cause a problem, since the default loader is unable to dynamically load user jobs; thus, the custom loader must be used. To use a custom loader for deserialization, a custom object input stream class is required that uses the custom loader to resolve the class instead of the default loader. Thus, the class `ObjectSpaceObjInputStream` is used to deserialize incoming object from the object space. This class is overkill for deserializing most incoming objects, but it is *required* to properly deserialize `ObjectSpaceStub` objects.

This section has shown one possible way to implement the Timed Function Execution pattern. By creating a stub function for each timed function and a corresponding class for each timed function that is executed in a separate thread, it is rather simple to implement this pattern. Additionally, adding a new timed function only requires adding a new stub method and creating the corresponding helper class. While implementation of the worker node object space may seem complex, thankfully the server implementation isn't nearly as complicated.

A.3 Implementing the Server Object Space

As mentioned in Section 4.4.4, handling incoming object space server requests from worker nodes on behalf of active objects is handled using the Reactor pattern. When an object-space job is first submitted to JOLTS, a single `ConcreteObjectSpace` is created and stored in a list of object spaces for that job (since object-space jobs invariably contain more than one active object, each of which has its own `self` object space). When an active object is executed on a worker node, the index of the `ConcreteObjectSpace` in the list of object spaces is stored in the `spaceID` field in the `ObjectSpaceStub` (*see* Figure A.4). When an `ObjectSpaceStub` object attempts to connect to the server, the connection request is received by the `ObjectSpaceHandler` object. The proper `ConcreteObjectSpace` target for the connection is located and a special `ChannelRecord` object is created to keep the incoming socket channel and `ConcreteObjectSpace` connected. The advantage of this setup is:

1. When any new data arrives on the channel, the system instantly knows what object space the request is for.

2. The channel is only created once for all requests for a specific object space, from a specific active object; i.e., the channel is left open between requests.

3. Multiple channels can easily be watched by a single multiplexor.

The one disadvantage of this setup is that if enough active objects exist on the system at once, there is a possibility that the server *might* run out of available ports. Once the `ChannelRecord` object is created, its associated channel is registered with the multiplex so it can be watched.

A.3.1 Object Space Multiplexor

When an object space request arrives, the socket channel is deregistered from the multiplexor and the type of request is determined (e.g., an `out` request) and the proper child object of `ObjectSpaceRequest` is created to handle the request, *see* Figure 4.14 on page 88. The child object is executed from inside a thread pool, which allows multiple requests to the same (or different) object space to be processed at the same time. The children objects of `ObjectSpaceRequest` are responsible for reading in the arguments for the requested operation, invoking the method on the object space, and returning the results through the channel. Recall that the target object space is attached to the invoking socket channel in a `ChannelRecord` object. This object is passed to the constructor of the object that will actually service the incoming object space request. The result of the operation is sent back through the channel regardless of the time required for the operation. The timing concerns are handled by the `ConcreteObjectSpace` instance discussed in Appendix A.3.2. Once the results have been sent back to the active object on the worker node, the channel must be reregistered with the multiplexor, which is not a simple process.

A multiplexor is supplied as part of the Java NIO package in the form of the `Selector` class. Channels can be watched for different state changes in the `Selector` object, the main ones of interest here are:

- `acceptable` — a new connection has arrived, used mainly on server channels

- `readable` — some data has arrived on the channel that need to be processed

The steps for handling `acceptable` are as follows:

1. The new channel is obtained.
2. The target object space is located (or created).
3. A `ChannelRecord` object is created to connect the object space and the channel.
4. The channel is registered with a `Selector` object to monitor it for entry into the `readable` state.

When a channel enters the `readable` state, it stays in that state as long as any data remains in the channel that can be read, *not* just if new data has arrived on the channel. This is not really a problem if the thread operating the multiplexor is also responsible for processing *all* the data from the channel, such as in the `FileStreamHandler` class discussed in Section 4.4.1. However, if data from the channel is processed by a different thread, as is the case in the `ObjectSpaceHandler` object, it can cause a serious problem if a channel isn't deregistered when it first entered the `readable` state.

If a channel that just entered the `readable` state isn't removed from the set of channels being monitored by the multiplexor, each time the multiplexor checks its monitored channel, it will see the same channel. This will cause an additional thread to start processing the data from the channel. The process will repeat until there is no data in the channel. Now multiple threads will be reading concurrently from the same channel, which is a *big* problem. Removing the channel from the set of channels being monitored by the multiplexor when it is first detected to be in the `readable` state solves the problem, resulting in only one thread processing the data in the channel. Once the thread used to process the data has finished executing (the channel is no longer in the `readable` state), the channel needs to be reregistered with the multiplexor so it can be watched until it enters the `readable` state again. The process of reregistering a channel with the multiplexor is non-trivial.

A very poorly documented feature is that the `Selector`/multiplexor has a lock on the set of channels it is watching, so a new channel can't just be added to the set blindly (it usually results in deadlock if tried). The multiplexor must be forced to release it's lock by performing an immediate selection of channels whose state have changed (the result is often zero when forced). Since a thread switch is required for this to take place, often the multiplexor releases and then re-obtains the lock its channel set before any new channels can be added. The solution is the two methods shown in Figure A.6. When an object space request finishes, it invokes the `reregister()` method. The `ChannelRecord` object is placed in a temporary list and the multiplexor is forced to release its lock by calling method `selector.wakeup()`. When the thread using the multiplexor wakes up, the first thing it does is add any pending channels to be registered to its set of watched channels by calling method `moveOverPendingRec()`. Now the multiplexor has the proper/complete set of channels it is supposed to be monitoring.

A.3.2 Implementing the `ConcreteObjectSpace` Class

As mentioned earlier, when an object-space operation request arrives, a descendant of class `ObjectSpaceRequest` is created to service the request. All the children instances (except

```
/** This method is used to reregister a channel with the selector. Because the thread watching
    the selector is most likely waiting, to get it successfully into the set being watched again, it needs
    to be placed in a list, then the selector woken up so it can check the list.
    @param chanRec The record containing the channel to be registered.
    @throws IOException Thrown if a problem occurs moving the channel back into non-blocking mode. */
protected synchronized void reregister(ChannelRecord chanRec) throws IOException
{
    chanRec.sockChannel.configureBlocking(false);
    toRegister.add(chanRec);
    selector.wakeup(); // force main thread to return from 'select' even if none available
}

/** This method is used to take all the channels waiting to be registered, and actually register
    them with the selector. Only the main thread, the one controlling the selector, calls this
    method. It needs to be synchronized to prevent more channels from being added while {@link
    #toRegister toRegister} is being emptied. */
private synchronized void moveOverPendingReg()
{
    while(!toRegister.isEmpty())
        try
        {
            ChannelRecord temp = (ChannelRecord) toRegister.removeFirst();
            temp.sockChannel.register(selector, SelectionKey.OP_READ, temp);
        }catch(ClosedChannelException e)
        {
            System.err.println("attempted to register a closed channel");
        }
}
```

Figure A.6: The two methods responsible for adding channels to the multiplexor

`EvalRequest`) operate on a specific instance of `ConcreteObjectSpace`. The first thing to realize when implementing this class is that it *must* be designed to handle concurrent access. The specific problem to be dealt with can be categorized as the standard multiple reader, multiple writers problem, with a few small additions. The main addition is the fact that both the readers and writers may enter and leave the same critical sections multiple times before they are complete. The following table classifies each of the available object-space operations.

Operation	reader	writer
rd	yes	
in		yes
out		yes
rm		yes
rmOut		yes
eval	N/A	N/A

As previously mentioned, the `eval()` method doesn't directly use the object space. Before any operation can begin, the thread must obtain the appropriate lock on the object space. Multiple readers can have concurrent access, while only one writer can be active at a time. Additional readers must wait if one (or more) writers are also waiting to gain the lock on the object space. No queues are needed to implement this functionality, only two counters and a Boolean. Manipulation of these three variables are only done inside critical sections. This is basically a primitive semaphore implementation, *without* using a

custom semaphore class. Future version of JOLTS may change to use an inner Semaphore class.

Once a request obtains the appropriate lock, it begins execution. Most of the operations require first finding the min/max number of matches to a supplied template. After *each* object in the object space is checked, the clock is also checked to make sure there is time remaining. If time has expired the results are calculated based on the matches found so far, the lock is released, and the function exists. If the maximum number of required matches are found, the same set of steps happen. Object space operations become interesting when all the items have been checked, but the maximum number of matches haven't been found *and* there is still time remaining. If this happens, the lock is released and the thread waits! Two different events can cause the waiting thread to wakeup:

1. The time for the operation has expired. In this case the matches already found are verified to still be present in the object space (they may have been removed by another thread). The results are determined using the remaining matches, the lock is released, and the results of the function is returned.

2. Some *other* thread has performed some type of write operation on the object space, either adding or removing objects. The awakened thread then starts checking the entire object space for a *new* set of matches, discarding the old set of matches. The whole process starts over again depending on whether or not the maximum number of matches were found.

Because so many operations on the object space require finding matches, this functionality was moved to a support method, given in Figure A.7. This is the second way to implement the Timed Function Execution pattern. As mentioned in Section 4.3.3.1, the looped implementation is only applicable to a small set of functions. Luckily, the object space primitives in `ConcreteObjectSpace` fall inside this set of applicable functions.

Using the `rd` function as an example, Figure A.8 contains the `rd()` function from inside the `ConcreteObjectSpace` class. As can be seen at the start of the loop, the first step is to call method `find()` to attempt to find the maximum number of matches. If either the maximum number of matches has been found or time has expired, the loop is exited, the lock released, and the function returns. If the maximum number hasn't been reached and time remains, the thread releases the lock and waits. When it wakes up, if the time remains the loop is restarted; otherwise, the method `removeStale()` is called to remove any stale objects in the set of matches. Most of the other object-space write operations are performed in a similar manner.

A rare occurrence can happen to methods `rm`, `rmOut`, `rd`, or `in` when it's timeout value is reached while executing method `timedWait()`, and some other thread has a write lock. When the awakened/expired thread finally obtains its lock, it is too late to do another pass through the object, since this can potentially be a long operation and the timeout value has already been reached. Instead, the matches that were found earlier are checked to see if they still exist in the object space, since they potentially could have been removed by a "write" thread that was executing when the current thread's time expired. Thus, the `removeStale()` method, shown in Figure A.7, is used to check each of the previously found matches to see if they still exist in the object space. If an object no longer exists, it is removed from the list of previously discovered matches. If the size of the list drops below the `min` required, the entire list is emptied. In the case that a "write" thread was *not* executing when the thread executing `timedWait()` awoke, all the awoken thread's

```
/** This method is responsible for searching through the object space for matches. So many of the
    object space primitives require this feature, this method was created to prevent code
    duplication in all the methods that require it. All the objects will be compared with
    <tt>obj</tt> using the {@link OILObject#match(OILObject) match} method. It will return right
    away either if the time has run out, or <tt>max</tt> matches have been found.
    @param obj The object that will be treated as a template to find matching objects.
    @param max The maximum number of matches required.
    @param timeout The amount of time for the entire object-space operation, <i>not</i> just this
        method.
    @param start The time when the entire object-space operation started, so it can be determined
        if the amount of time allowed has expired. */
private Vector find(OILObject obj, int max, int timeout, long start)
{
    Vector tempVec = new Vector();
    int size = container.size();
    for(int i = 0; i < size; i++)
    {
        OILObject temp = (OILObject) container.get(i);
        if(obj.match(temp))
        {
            tempVec.add(temp);
            if(tempVec.size() == max)
                return tempVec;
        }
        if(!timeLeft(start, timeout))
            return tempVec;
    }
    return tempVec;
}

/** When a search operation times out, it is possible that matches previously found have been
    removed by some other thread while the operation was waiting. This method is responsible
    for taking those results, and checking to make sure they are all still here. This method
    will remove any entries in <tt>vec</tt> that are no longer in the object space. If the
    number of remaining entries is below <tt>min</tt>, they will all be removed.
    @param vec The matches to check to make sure they aren't stale. It is possible this set
        will be emptied completely by this method.
    @param min The minimum number of items allowed in <tt>vec</tt>.
    @see java.util.Vector#containsAll
    @see java.util.Vector#contains */
private void removeStale(Vector vec, int min)
{
    if(vec.size() < min) // do we even have the minimum?
        vec.clear();
    else if(!container.containsAll(vec)) // need to eliminate the ones missing
    {
        for(int i = 0; i < vec.size(); i++) // can't use a size variable optimization
            if(!container.contains(vec.get(i)))
                vec.remove(i--);
        if(vec.size() < min) // do we have enough remaining objects after removing stale objects?
            vec.clear();
    }
}
```

Figure A.7: Two support methods from `ConcreteObjectSpace`

```
/** This method implementation is identical to {@link #in} except that the matches found
    aren't removed from the object space, and it works on a read lock instead of a write lock. */
public MultiSet rd(OILObject obj, int min, int max, int timeout)
{
    Vector tempVec = new Vector();
    long start = System.currentTimeMillis();

    obtainReadLock();
    while(true) // intentional infinite loop
    {
        tempVec = find(obj, max, timeout, start);
        if(tempVec.size() == max)
            break;
        if(!timeLeft(start, timeout))
        {
            if(tempVec.size() < min)
                tempVec.clear();
            break;
        }

        releaseReadLock();
        timedWait(start, timeout); // time to sleep
        obtainReadLock();

        if(timeLeft(start, timeout)) // woken by someone doing a write operation
            continue;

        removeStale(tempVec, min);
        break;
    }
    releaseReadLock();
    return vectorToMultiSet(tempVec);
}
```

Figure A.8: The `rd` function in `ConcreteObjectSpace`

previous matches will still be in the object space, and the call to method `removeStale()` isn't required. However, it is impossible to determine which thread(s) were executing when the time expired inside method `timedWait()`; thus, method `removeStale()` must always be invoked.

A.4 Implementing `CheckpointMech`

As mentioned in Section 4.3.2 on page 74, the actual checkpoint mechanism passed to user jobs is an instance of (or descendant) of class `ExecuteNewGridJob`, shown in the left-hand branch of Figure A.9. Depending on the type of incoming job, the appropriate class from the `ExecuteNewGridJob` hierarchy is used to begin servicing the job.

1. `ExecuteNewGridJob` — a class used to execute *new* simple/sequential jobs, and MISD sub-jobs.

2. `ExecuteNewSIMDJob` — a class used to execute new SIMD sub-jobs. It is much more complicated than the previous class because it needs to download the parameters from the server, and find the proper constructor for the sub-job before it can create it and begin executing the sub-job.

3. `ExecuteObjectSpaceJob` — a class used to for executing active objects (*see* Section 4.3.3).

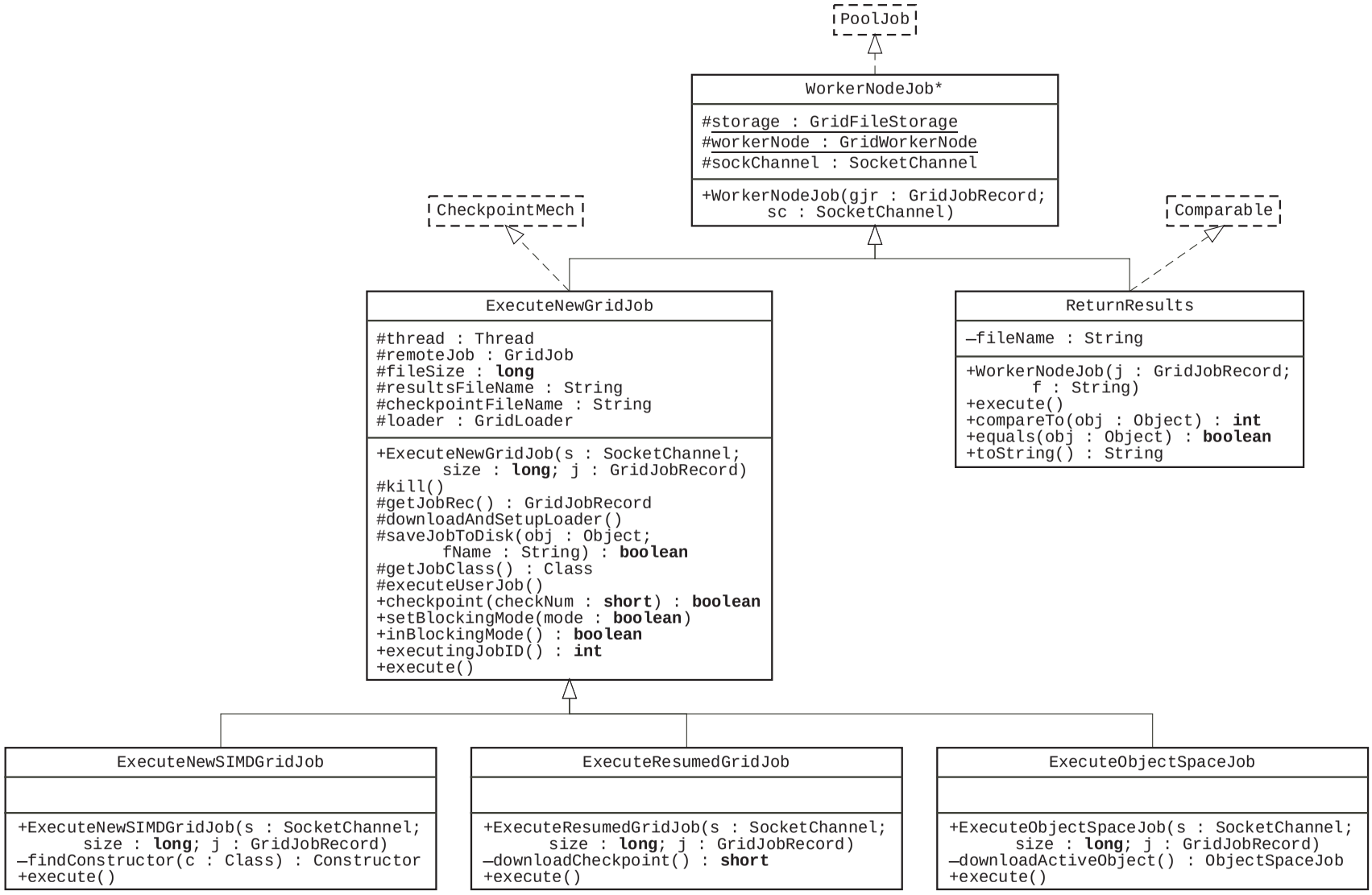

Figure A.9: The `WorkerNodeJob` hierarchy

4. `ExecteResumedGridJob` — a class used when *resuming* any of the four jobs types from a checkpoint file.

Only class `ExecuteNewGridJob` contains the method for creating checkpoints, but it is inherited by all its descendants. The checkpoint mechanism that is passed to the user's job by way of the `setCheckpointMech()` method must be declared as **`transient`** because as can plainly be seen in Figure A.9, class `ExecuteNewGridJob` and its descendants are *not* serializable. It would be possible to declare the class as serializable, and then make the appropriate fields transient; e.g., `thread` would have to be transient. This would mean the checkpoint mechanism stored in the user's program wouldn't have to be transient.

There is no real advantage to doing this, since a new checkpoint mechanism would still have to be passed to the user's job when it is deserialized on a worker node. It is much easier to check to ensure that checkpoint mechanism is declared **`transient`** in the user's program (*see* Appendix A.1), than it would be to properly deserialize an instance of `ExecuteNewGridJob` properly. To do this would require a *lot* of typecasting to obtain values out of the current thread and assign them to the deserialized object.

Appendix B

Experimental Data

This appendix contains the experimental data for some of the experiments run in Chapter 5. The data for the server stress test experiment is *not* given. The following is the data for Figure 5.3 on page 98, all times are in seconds.

Threads/ Nodes	Standalone	Thread Ideal	Threaded Local	Blocks = Nodes	Blocks = 400
1	7647.67	7647.67	7647.67	5540.78	7417.33
2	7647.67	3823.84	5643.13	3954.99	4044.88
3	7647.67	2549.22	5382.38	2639.54	2693.86
4	7647.67	1911.92	7366.91	1973.93	2012.54
5	7647.67	1529.53	7020.53	1577.01	1612.51
6	7647.67	1274.61	6766.35	1312.59	1348.15
7	7647.67	1092.52	7721.25	1128.16	1151.21
8	7647.67	955.96	7457.45	989.49	1008.05
9	7647.67	849.74	7471.05	905.45	906.87
10	7647.67	764.77	7427.60	823.69	807.54
11	7647.67	695.24	7797.64	743.44	743.34
12	7647.67	637.31	7562.77	674.20	683.23
13	7647.67	588.28	7458.63	650.83	625.69
14	7647.67	546.26	7991.32	613.60	587.34
15	7647.67	509.84	8088.54	569.98	547.87
16	7647.67	477.98	7652.54	527.38	507.79
17	7647.67	449.86	8100.99	498.31	484.56
18	7647.67	424.87	8240.18	474.41	464.63
19	7647.67	402.51	7864.62	443.27	430.78
20	7647.67	382.38	7937.26	427.05	407.47
21	7647.67	364.17	8465.49	393.46	387.50
22	7647.67	347.62	8017.88	375.88	382.61
23	7647.67	332.51	7990.22	359.54	364.93
24	7647.67	318.65	8210.01	344.91	351.17
25	7647.67	305.91	8154.70	331.27	326.71
26	7647.67	294.14	8112.11	318.33	325.48
27	7647.67	283.25	8359.89	307.11	305.06
28	7647.67	273.13	8567.85	296.09	303.84
29	7647.67	263.71	8204.17	286.51	285.00
30	7647.67	254.92	8166.06	277.11	284.18
31	7647.67	246.70	8400.90	268.80	265.86
32	7647.67	238.99	8544.64	260.74	267.02
33	7647.67	231.75	8553.05	253.21	263.14

Threads/ Nodes	Standalone	Thread Ideal	Threaded Local	Blocks = Nodes	Blocks = 400
34	7647.67	224.93	8535.04	245.84	245.92
35	7647.67	218.50	8554.72	239.29	245.48
36	7647.67	212.44	8430.27	233.07	242.45
37	7647.67	206.69	8655.58	226.92	227.00
38	7647.67	201.25	8651.49	220.91	226.07
39	7647.67	196.09	8845.89	215.58	223.76
40	7647.67	191.19	8622.20	210.30	208.29
41	7647.67	186.53	8991.68	205.59	205.69
42	7647.67	182.09	8772.19	200.67	207.62
43	7647.67	177.85	8914.38	196.17	206.81
44	7647.67	173.81	8971.25	191.77	204.33
45	7647.67	169.95	8933.83	187.71	190.38
46	7647.67	166.25	8855.60	183.79	189.38
47	7647.67	162.72	9072.92	179.97	189.81
48	7647.67	159.33	8947.87	176.41	188.82
49	7647.67	156.07	8977.68	172.64	185.69
50	7647.67	152.95	8955.72	169.38	169.51
51	7647.67	149.95	9094.27	165.85	172.68
52	7647.67	147.07	9036.48	162.86	172.27
53	7647.67	144.30	8997.21	159.95	168.48
54	7647.67	141.62	9180.27	157.20	171.60
55	7647.67	139.05	9210.01	154.38	171.30
56	7647.67	136.57	9190.36	151.75	167.16
57	7647.67	134.17	9361.10	149.40	153.04
58	7647.67	131.86	9361.72	146.85	157.21
59	7647.67	129.62	9236.08	144.68	154.91
60	7647.67	127.46	9314.24	142.26	153.36
61	7647.67	125.37	9826.14	140.03	151.72
62	7647.67	123.35	9825.68	138.02	150.47
63	7647.67	121.39	9482.70	135.82	148.72
64	7647.67	119.49	9866.74	133.81	150.33

The following is the data for Figure 5.6 on page 105, all times are in seconds.

Threads	Thread Ideal	Stand Alone Blocks = nodes	Stand Alone Blocks = 400
1	5395.73	5339.36	5545.93
2	2697.87	5401.21	5595.74
3	1798.58	5473.96	5659.28
4	1348.93	7442.21	5681.44
5	1079.15	7094.31	5614.57
6	899.29	6813.51	5682.85
7	770.82	7797.43	5680.82
8	674.47	7566.59	5676.11
9	599.53	7385.93	5862.46
10	539.57	7253.74	5904.31
11	490.52	7910.69	5783.41
12	449.64	7728.56	5956.78
13	415.06	7560.83	5922.11
14	385.41	8151.66	5957.64

Threads	Thread Ideal	Stand Alone Blocks = nodes	Stand Alone Blocks = 400
15	359.72	7909.19	5822.85
16	337.23	7794.73	6066.34
17	317.40	8195.70	6093.31
18	299.76	8083.36	5965.67
19	283.99	8030.35	6066.39
20	269.79	7958.03	6341.66
21	256.94	8334.82	6271.65
22	245.26	8155.91	6149.60
23	234.60	8154.14	6479.36
24	224.82	8460.81	6261.77
25	215.83	8441.11	6506.82
26	207.53	8408.95	6246.40
27	199.84	8565.29	6493.03
28	192.70	8531.08	6755.89
29	186.06	8461.65	6293.34
30	179.86	8452.05	6544.62
31	174.06	8577.86	6699.68
32	168.62	8602.60	6426.05
33	163.51	8538.23	6664.84
34	158.70	8827.81	6756.36
35	154.16	8735.35	7115.73
36	149.88	8682.44	6413.40
37	145.83	8924.65	6601.33
38	141.99	8948.43	6948.61
39	138.35	8883.84	7147.91
40	134.89	8785.77	7743.59
41	131.60	9012.35	6684.05
42	128.47	8991.62	7404.07
43	125.48	9029.68	6989.83
44	122.63	9188.43	7220.49
45	119.91	9141.07	7383.61
46	117.30	9115.38	7590.08
47	114.80	9363.87	6749.42
48	112.41	9247.31	6757.81
49	110.12	9272.45	7047.29
50	107.91	9269.99	7329.79
51	105.80	9518.26	7442.84
52	103.76	9449.89	8025.54
53	101.81	9401.11	8104.10
54	99.92	9551.42	8143.09
55	98.10	9594.31	8002.62
56	96.35	9533.07	8125.44
57	94.66	9712.64	7035.76
58	93.03	9671.02	7192.67
59	91.45	9624.50	7363.31
60	89.93	9628.12	7429.52
61	88.45	9745.61	7567.75
62	87.03	9756.70	7748.46
63	85.65	9736.10	7989.58
64	84.31	9949.47	8051.68

Threads	On Grid Blocks = nodes	On Grid Blocks = 400 No Checkpoints	On Grid Blocks = 400 Checkpoints
1	5486.74	5452.09	5507.99
2	3888.14	2747.17	2767.11
3	2651.47	1856.19	1845.13
4	2006.68	1398.73	1398.49
5	1601.09	1125.20	1124.01
6	1341.20	943.77	937.04
7	1140.95	812.08	812.74
8	1004.85	713.48	714.62
9	884.85	635.99	634.91
10	802.82	582.35	575.52
11	749.50	528.00	523.18
12	670.31	485.35	483.02
13	623.81	447.63	451.10
14	572.66	423.69	420.01
15	539.15	388.38	387.41
16	501.50	366.43	366.10
17	475.96	345.68	343.04
18	454.69	329.92	322.90
19	431.42	305.39	303.53
20	406.59	303.90	300.86
21	395.40	289.36	283.91
22	385.32	275.00	264.55
23	360.70	268.74	262.44
24	357.97	253.68	247.93
25	326.86	245.25	245.55
26	314.02	229.92	227.08
27	307.64	225.14	225.38
28	293.50	225.36	225.05
29	282.70	212.05	207.35
30	274.31	206.99	205.88
31	261.95	205.85	205.91
32	255.36	188.32	187.69
33	249.30	189.34	186.81
34	243.53	186.51	184.96
35	234.45	188.18	185.55
36	229.98	174.15	168.77
37	226.47	168.35	167.76
38	218.50	167.17	166.55
39	216.54	167.48	165.40
40	208.55	168.73	166.01
41	203.23	147.20	146.61
42	195.90	148.28	146.61
43	191.53	149.04	147.32
44	187.75	145.78	146.18
45	187.72	147.56	146.24
46	183.70	149.50	146.19
47	181.23	128.17	131.89
48	176.75	128.49	131.47
49	172.68	127.77	131.43
50	169.99	131.88	131.44

Threads	On Grid Blocks = nodes	On Grid Blocks = 400 No Checkpoints	On Grid Blocks = 400 Checkpoints
51	166.25	131.44	127.13
52	159.88	130.36	128.75
53	159.20	129.50	126.87
54	154.64	129.85	126.27
55	153.13	128.81	130.19
56	150.52	126.85	125.96
57	148.76	107.35	108.94
58	147.11	107.12	108.80
59	142.88	108.87	107.77
60	142.53	109.33	108.32
61	140.87	109.86	106.28
62	137.82	107.40	110.86
63	133.64	107.44	111.17
64	134.43	107.51	110.18

Appendix C

Glossary

ADT Abstract Data Type — An abstract specification of a set of values and associated operations on those values.

ALU Arithmetic Logic Units — part of the CPU responsible for performing arithmetic and logic operations.

API Application Programmer's Interface — a set of functions/procedures that a program either needs to implement or use to tie into an existing system.

CAVE A recursive acronym, CAVE Automatic Virtual Environment. The CAVE is a projection-based VR display, where the user stands in the center of display area.

CLI Command Line Interface — a text-only way of interacting with a program.

CORBA Common Object Request Broker Architecture — a protocol used to help perform RPC between multiple languages.

DSM Distributed Shared Memory — a technique used to manage memory on multiple computers. It can be implemented using either hardware or software.

GUI Graphical User Interface — a visual way of interacting with a program using windows, buttons, menus, and so on.

HPC High Performance Computing — an area of computer science that grid computing started from.

HPF High Performance FORTRAN — a common language which is used to implement large scale concurrent systems.

IPC Inter-Process Communication — processes are able to communicate with each other using either message passing or shared memory.

JOLTS Java Objective Linda Tuple Space — a Java-based grid system that uses the Objective Linda specification for communications between multiple sub-jobs.

JVM Java Virtual Machine — a stack based virtual machine capable of executing Java bytecode.

K Kilobyte — a unit of measure consisting of 1024 *bytes*.

Kb Kilobit — a unit of measure consisting of 1024 *bits*, used primarily when talking about network speeds.

M Megabyte — a unit of measure consisting of 1024 Kilobytes.

Mb Megabit — a unit of measure consisting of 1024 Kilobits, used primarily when talking about network speeds.

MISD Multiple Instruction Single Data — different instructions repeated using the same data set. Often also referred to Multiple *Program* Multiple Data.

MPP Massively Parallel Processor — a computer with a large number of processors.

OIL Object Interchange Language — a language-independent notation for describing ADTs in Objective Linda.

ORB Object Request Broker — a middleware technology that manages communication and data exchange between objects.

RMI Remote Method Invocation — a Java specific version of RPC.

RPC Remote Procedure Call — short form for invoking functions/procedures on non-local resources.

SIMD Single Instruction Multiple Data — the same set of instructions that is repeated using different data sets. It is often also referred to as Single *Program* Multiple Data.

Switchboard An intermediary between an open socket connection and a remote computer. Used to help keep socket connections alive when a program migrates between computers.

TCP/IP Transmission Control Protocol Internet Protocol — a set of state-based protocols used by computers to communicate with each other over the Internet.

UDP User Datagram Protocol — a stateless protocol used to transmit data packets between computers.

XML Extensible Markup Language — a flexible "meta" language used for defining markup languages.